The Intermediate Language for the DPCE compiler

P.J. Hatcher and Daniel Luchaup, University of New Hampshire

March 7, 1996

1 The basic components of a DPCE compiler.

In this document we make the assumption that the reader is familiar with the specifications of the DPCE language.

The DPCE compiler consists of two separate components : the Front End (FE) and the Back End (BE). The FE takes as input a file containing a DPCE program and outputs a file containing the Intermediate Language (IL) description of the DPCE program. Subsequently, the BE processes the IL file and generates a file containing the proper description of the original program, for the underlying hardware. The last file may be either a C file or an assembly language file. From the FE's point of view, the IL is intended for an *Abstract Parallel Machine* (APM), on which the program will run.

2 Language Description.

Here are some of the issues for the IL :

1. The IL program has to preserve the semantics of the original DPCE program.
2. The IL program has to have a printable ASCII form.
3. The APM may run on several different machines, so the IL is not allowed to make any assumption about the underlying hardware. This means that the IL may represent a simplified version of the C language (such as a three address intermediate language), but is not allowed to entirely change the parallel instructions that may be specific to the hardware. For example, the FE cannot assume that the program will run on a SIMD hardware, since in reality the underlying hardware may be a Local Area Network.
4. Since DPCE and C* are closely related, the IL should be able to be used as an intermediate language for C* as well.

IL is a block-structured language. With few exceptions, its form is that of a three address code. In general, IL identifiers represent addresses. An object has to be dereferenced if the value at its address is needed. However, in order to be able to get the right data from that address, the type of the object is required too. Thus, additional information is required for each IL object (and operator)[1].

An IL file consists of declarations and definitions of variables and functions.

[1]For a detailed description of an IL object see the *Basic IL objects* subsection

In the grammar of IL, we represent terminals using only capital letters. Since many of the nonterminals are common to those in DPCE, we do not further expand those nonterminals in our description (this is true **only** for our presentation, but in the real IL grammar these nonterminals are expanded as well). The distinction between the DPCE nonterminals and the one specific to IL is that the name of a nonterminal specific to IL starts with the IL prefix. Ex. "ILParam" is specific to IL, whereas "identifier" is borrowed from the DPCE grammar.

If IL is translated back to C, we denote the resulting program as a *C2 code*.

The file name and line of the file are necessary for error reporting, debugging or performance analysis. This information is given for any statement or declaration of the IL in the following form:

```
ILLocation :
                IL_FILE_NAME IL_LINE_NUMBER IL_COLUMN_NUMBER
        ;
```

The IL_LINE_NUMBER and IL_COLUMN_NUMBER are the line number and the column number of the position in the file IL_FILE_NAME where the DPCE grammar construct was encountered.

2.1 Declarations.

IL can declare an object, a structure, an union, an enumeration type, or a function.

```
ILdeclaration :
ILLocation ILdeclaration_rest
        ;

ILdeclaration_rest :
                ILObject_Decl
        |       ILStruct_or_Union_Decl
        |       ILEnum_Decl
        |       ILTypedef
        |       ILFunction_Declaration
        ;
```

2.2 Objects.

Objects have fixed size. Memory is allocated for them, unless the "extern" specifier is present.[2]

```
ILObject_Decl           :
                OBJECT  identifier      storage_class_specifier type_name
                        ILInitializer
                ENDOBJECT
        ;


ILInitializer :
```

[2] Due to shape−array ambiguity in type_name, this section may change

```
                ILValue_list
        |       ILstatement
        ;

ILValue_list :
                /* empty */
        |       ILValue_list
                VALUE ILConstant

        ;
```

If an object is not extern, it may have an initializer. The initializer may contain code to compute the initial values. The value is assigned to the object using a VALUE statement. If a function call is needed to initialize the object, then a compound statement is used inside the initializer. This is done because the function call may require some temporaries which can be declared in the compound statement. The name of the object is *visible* inside the compound statement, so that it can be assigned the resulting initial value.

The VALUE initializers are used for structure components. The initial values of the components are assigned in order. Note that "storage_class_specifier" and "type_name" are the same as in DPCE. However "type_name" must not be a function type. Functions are not objects (they don't have a fixed size).

Any type component contained in "type_name" should be either a type name, or a TYPEDEFed name. This mechanism is used to solve the problems that occur when a non-constant expression is used as a shape.

```
Ex.:    The syntacticly correct DPCE declaration
        ``int:(f(x)) * : (g(y)) p ; ''  is translated to :

        OBJECT  dummy_name1     static SHAPE
        ... code to compute f(x) ...
        VALUE <result of f(x)>
        ENDOBJECT
        OBJECT  dummy_name2     static SHAPE
        ... code to compute g(y) ...
        VALUE <result of g(y)>
        ENDOBJECT
        OBJECT  p       automatic       int:dummy_name1 (*:dummy_name2)
        ENDOBJECT
```

In this case, the non-constant shape values f(x) and g(y) are considered as initializers for some temporary shape objects. These temporaries are initialized prior to their use in the declaration of *p*. In this way, when *p* is declared, all the components of its type are completely determined.

2.3 Basic IL objects.

The definition of an IL variable is used to determine the space requirements for that variable. After that the BE will work mostly with IL objects. An ILobject specifies the data which is manipulated by the IL. This data consists either of a constant or of a variable.

```
ILobj :
                ILobject_specifier identifier
        |       ILobject_specifier ILConstant
        ;

ILConstant :
                INT_CONSTANT | DOUBLE_CONSTANT | STRING_LITERAL
        ;

ILobject_specifier :
        ILoperator_spec DEREF | ILoperator_spec
        ;

ILoperator_spec :
        `('ILparallel_kind ILtype `)'
        ;

ILparallel_kind :
                NON_PARALLEL        | PARALLEL
        ;

ILtype : /* to be specified */
                ILsimple_type
        |        ILpointer_type
        ;

ILpointer_type :
                POINTER2_PARALLEL ILsimple_type
        |       POINTER2_NON_PARALLEL ILsimple_type
        ;
```

Usually, IL interprets an identifier as an address but in most cases the data at that address is required. An IL objects consists of an IL variable (the *identifier*) or constant and additional information (ILobject_specifier) about how to interpret the memory content stored at the address given by that variable or constant.

This information may tell to the BE whether it needs the address or the contents (DEREF) of the variable. The parallel/non-parallel nature of the data is determined by ILparallel_kind. Finally, the type of the data is specified by ILtype. The last two specifiers are given for most of the operators. In this case, ILoperator_spec determines the type of the DEREF operator (or, an implicit NULL operator whenever just the address of the IL object is needed).

2.4 Structures and Unions.

```
ILStruct_or_Union_Decl :
                ILStruct_Decl
        |        ILUnion_Decl
        ;

ILStruct_Decl :
                FORWARD STRUCT identifier
        |        STRUCT identifier
                        ILStruct_declaration_list
                ENDSTRUCT
        ;

ILUnion_Decl :
                FORWARD UNION identifier
        |        UNION identifier
                        ILStruct_declaration_list
                ENDUNION
        ;

ILStruct_declaration_list :
                ILStruct_member
        |        ILStruct_member ILStruct_declaration_list
        ;

ILStruct_member :
                identifier type_name
                identifier type_name INT_CONSTANT /* bit fields */
        ;
```

The layout of the components in a structure or a union is the same as in the C language. The original names of the structure and its components are preserved.

The forward declaration of a structure is intended only for the translation from IL to C.

Any nested structures / unions are (recursively) unnested. Unnamed structures/unions/enums will get a name.

```
        Ex.:
                struct { int x,y;
                         struct { float r, a;} par;
                        } point;
                is translated to :
                STRUCT dummy_struct_name1
                        r       float
                        a       float
                ENDSTRUCT
                STRUCT dummy_struct_name2
```

```
                x       int
                y       int
                par     struct dummy_struct_name1
        ENDSTRUCT
        OBJECT  par     automatic       struct dummy_struct_name2
        ENDOBJECT
```

2.5 Functions declarations.

A function declaration contains the function name, return type and a list of abstract parameters.

```
ILFunction_Declaration :
                FUNCTION_DECL ILFunction_spec identifier
                RETURNS type_name
                ILAbstract_param_list
                END_FUNCTION_DECL
        ;

ILAbstract_param        :
                ILParam
        |       type_name
        ;
ILParam                 :
                identifier      type_name
        ;
```

The function specifier **function_spec** specifies whether the function is elemental, nodal or neither. It also gives information about the prototype of the function (old style declaration, variable number of arguments or a prototyped declaration).

```
ILFunction_spec :
                '(' IL_C_spec IL_PARALLEL_spec ')'
        ;

IL_C_spec :
                C_OLD_STYLE | C_DOTS | C_PROTOTYPE
        ;

IL_PARALLEL_spec :
                ELEMENTAL | NODAL | SIMPLE
        ;
```

2.6 Typedef and types.

A type definition introduces a new name for a type:

```
ILTypedef               :
                TYPEDEF identifier      type_name
        ;
```

Note that the types may be parallel or non-parallel. The non-parallel types are similar to those in C, except for pointers to parallel, arrays of parallel, and aggregate types which have a parallel element. A parallel type is specified by a shape and a non-parallel type. The shape itself is an IL (and also a DPCE) object which can be the result type of a function call. Thus, the returned value of a function can be used to determine a type. However, when the BE finds an object definition, the type of that object has to be completely determined, and cannot contain function calls. This is why we require that a temporary shape, which holds the result of the function call, be used in such cases.

3 Statements.

The statements in IL are the following :

```
ILStatement :
        ILLocation ILStatement_rest
        ;

ILStatement_rest :
                ILlabeled_statement
        |       ILcompound_statement
        |       ILexpression_statement
        |       ILselection_statement
        |       ILjump_statement
        |       ILextra_statement
        ;
```

Note that, if compared to C, there is no *ILiteration_statement*. The FE replaces all iterations with GOTO statements. However, it is the FE's job to generate additional information which may be used by the BE to reconstruct the original loops.

3.1 Labeled statements.

```
ILlabeled_statement     :
                identifier              ':'     ILStatement
        |       CASE INT_CONSTANT       ':'     ILStatement
        |       DEFAULT                 ':'     ILStatement
        ;
```

Label statements are similar to those in C.

3.2 Compound statements.

```
ILcompound_statement :
```

```
        BEGIN       /* change this if ambiguous !*/
        ILDeclaration_list
        ILStatement_list
        END
        ;
```

A compound statement contains a list of declarations and a list of statements. The new names introduced by the declaration are visible only inside the block of the statement (determined by the BEGIN and END keywords). The semantics are similar to that in C. For instance, if a jump statement outside the block has a target inside the block and if the block has some initialized local variables, then the behavior of the program is unspecified.

3.3 Selection statement.

```
ILselection_statement :
                SWITCH '(' ILobj ')'
                        ILStatement
        ;
```

The form of the selection statement is preserved as in C for two reasons. First: this decision will facilitate a translation from IL back to C. Second: in this way, the FE doesn't have to decide what kind of code to generate (GOTOs to labels stored in a table, or a sequence of IF statements).

3.4 Jump statement.

```
ILjump_statement :
                                        GOTO identifier
        |       IF_TRUE ILobj GOTO identifier
        |       IF_FALSE ILobj GOTO identifier
        |       BREAK
        |       RETURN
        |       RETURN ILobj
        ;
```

IL supports unconditional and conditional jumps, and return statements. In addition to these basic jump statements, it also supports the *break* statement. This is necessary because the form of the *selection_statement* is the same as in C.

The *return* statements are used to return from functions calls. If an ILobj is specified, then it contains the returned value of the function.

Note that there are no *continue* statements. They are replaced by *goto* statements. However, the FE may need to generate some extra information to allow the reader of the C2 code to understand the semantics of the original DPCE program.

3.5 Expression statements.

```
ILexpression_statement :
                ILinc_or_dec
        |       ILassignment_expr
        |       ILfunction_call
        |       ILparameter
        |       ILCompound_statement
        ;
```

These are the statements that may change the content of the IL variables. We could have replaced the entire set of C operations with a restricted set of operations. We decided not to do this in order to allow the BE to benefit from the operation set of different hardwares, and also to improve the readability of the C2 code. However, there are some exceptions. For instance, the && and || operators have a natural translation in the IL, using flow control, and are therefore no longer needed.

ILinc_or_dec describe the increment or decrement of an IL variable.

3.6 Assignment expressions.

These are the statements that can modify the IL objects:

```
ILassignment_stmt       :
                ILcopy_assign
        |       ILunary_assign
        |       ILbinary_assign
        |       ILcommunication
        ;
```

3.7 Copy statement.

The *copy_statement* is the simplest assignment statement.

```
ILcopy_assign :
                ILobj IL_asgn ILobj
        ;

IL_asgn :
                ILoperator_spec '='
        ;
```

The right ILobj specifies the data to be transfered. This data may be either the content of the variable mentioned in ILobject, if it is DEREFed, or the address of that variable, if it is not DEREFed. The data is of the type, and parallel kind given by the ILobject_specifier of the right ILobj. The assignment stores the data either into the variable specified by the left ILobj, if this is not DEREFed, or into the memory location pointed by the variable on the left.

3.8 Unary assignment.

This assignment assigns the result of a unary operator to an ILobj.

```
ILunary_assign :
                ILobj IL_asgn ILoperator_spec ILunary_op ILobj
                ILobj IL_asgn SHAPEOF ILobj
        |       ILobj ILoperator_spec ILreduction_op ILobj
        ;
```

The ILunary_op may be any unary operator in C, except for the **sizeof** operator.

If the operator is the SHAPEOF operator, then the right ILobj must denote a shape object. If the operator is a reduction operator then the right ILobj must be parallel, and the left ILobj must be non-parallel.

3.9 Binary assignment.

```
ILbinary_assign :
                ILobj IL_asgn ILobj ILoperator_spec ILbinary_op ILobj
        |       ILobj IL_asgn ILobj '['ILobj']' INT_CONSTANT
        |       ILobj IL_asgn ILobj '.' identifier INT_CONSTANT
        ;
```

The ILbinary_op may be any binary operator in C except && and ||.

For the binary, array index operator, we also provide an integer constant which tells the BE the offset of the data to be accessed. In fact, this is the only information needed for the BE to generate correct code. The ILobj is intended solely for a more convenient translation to C2 code.

Similarly, the structure/union dot operator only needs the INT_CONST offset and the base address given by the right ILobj. However, a convenient translation to C2 code can be done if we provide the extra information concerning the name (*identifier*) of the accessed field.

4 Communication.

Communication is a form of assignment in which different processing elements exchange data.

```
ILcommunication_assign :
                ILslice_assign
        |       ILparallelIndex_assign
        |       ILnonparallelIndex_assign
        |       ILgrid_assign
        ;
```

4.1 Slicing.

A slicing assignment in DPCE is translated using :

```
ILslice_assign :
                SLICE`['ILobj`]'ILobj IL_asgn ILobj
        |       ILobj IL_asgn SLICE`['ILobj`]' ILobj
        ;
```

The ILobj used as a index must be a two dimensional integer array whose number of rows equals the rank of the shape of the indexed ILobj. For each row (dimension of the right ILobj), it specifies the first value, the last value and the step for the slice.

4.2 Parallel indexing.

```
ILparallelIndex_assign :
                P_INDEX`['ILobj`]'ILobj IL_asgn ILobj
        |       ILobj IL_asgn P_INDEX`['ILobj`]' ILobj
        ;
```

In this case, the index ILobj is an array of parallel integers. Its shape is given by the shape of the parallel indices. The array has one element for each dimension of the indexed object. An entry in the array contains the parallel value used as a index for the corresponding dimension.

Observations

- The main assumption made by the front end is that the slice and parallel index expressions are used only for direct (via an assignment) data transfer. If the sliced/indexed expression is passed as an argument to a function, or if its address is assigned to a parallel pointer, the behavior is undefined.
- In this case, the code generated by the FE is not very different from the original DPCE code. This is because the FE doesn't know the nature of the communication primitives supported by the underlying hardware.

4.3 Non-parallel indexing.

```
ILnonparallelIndex_assign :
                NP_INDEX`['ILobj`]'ILobj IL_asgn ILobj
        |       ILobj IL_asgn NP_INDEX`['ILobj`]' ILobj
        ;
```

The ILobj used as index must be a non-parallel unidimensional integer array. The number of elements of the array must be equal to the rank of the indexed ILobj. Each entry in the array contains the non-parallel index used for the corresponding position of the parallel object.

4.4 Grid operations.

Grid communication is a particular form of parallel indexing.

```
ILgrid_assign :
                GRID`['ILobj`]'ILobj IL_asgn ILobj
        |       ILobj IL_asgn GRID`['ILobj`]' ILobj
        ;
```

The ILobj used as index must be a non-parallel unidimensional integer array. The number of elements of the array must be equal to the rank of the indexed ILobj. This is similar to non-parallel indexing, however the semantic is different. Each entry in the array contains the non-parallel index used as a grid index for the corresponding position of the parallel object.

5 Function calls and parameter passing.

```
ILparameter :
                PARAM ILobj
        ;
ILfunction_call :
                CALL ILFunction_spec identifier INT_CONSTANT
        |       ILobj CALL ILFunction_spec identifier INT_CONSTANT
        |         /* pointers to functions : */
                CALL ILFunction_spec DEREF identifier INT_CONSTANT
        |       ILobj   CALL ILFunction_spec  DEREF identifier INT_CONSTANT
        ;
```

Parameters are passed to functions using the *parameter* statement. It is up to the BE to handle these parameters. A parameter is an ILobj, and any expressions or function calls are replaced by the FE with temporary variables, which contain the result of the function call or expression evaluation.
There are two kinds of function calls: procedure call and function call. For function calls (when the return value is assigned to some object) the object which is to get the result of the function call is also specified. The INT_CONSTANT specifies the number of parameters for the corresponding function call. This is not redundant information, since function calls may be nested.

6 Current Shape control.

Every statement which operates on parallel objects executes in the context of a shape named the *current shape*. If the statement involves only nonparallel variables then the current shape is not really needed. If the statement is an expression statement then the shape of the statement becomes the current shape. Any ILobj which is parallel is considered to have the shape equal to the current shape.

```
ILShape_control :
                ILShape_selection
        |       ILShape_update
        ;

ILShape_selection :
                SELECT_SHAPE ILobj
```

```
        ;

ILShape_update          :
                PUSH_AND_SET ILobj
        |       PUSH_ALL_TRUE
        |       POP ILobj
        |       TOP ILobj
        ;
```

The ILShape_selection statement establishes a new current shape. All of the objects involved in a shape update are of parallel integer type, and their shape is the current shape.

ILShape_update statements are used to change the context of the current shape. They correspond to **where** and **everywhere** statements in DPCE. Since these statements may be nested, any shape has a stack of contexts. The top of the stack is the current context for that shape. Thus, the current context of a shape can be modified either by pushing or popping to or from the stack. For an *everywhere* statement it pushes a boolean whose values are all set to TRUE.

7 Extra statements.

IL provides an escape for embedding additional information such as dataflow analysis. This is done by extra-blocks, which are delimited by BEGIN_EXTRA and END_EXTRA keywords. Neither the FE or IL care about the information contained in these blocks. They are intended for the use of additional tools that may need to process the IL. The only restriction IL imposes is that these blocks are placed only in positions where statements can occur.

```
ILextra_statement :
                BEGIN_EXTRA
                        .....
                END_EXTRA
        ;
```