A Remote Procedure Call Implementation for the Cell Broadband Architecture

1 Introduction

In this paper, we present an implementation of Remote Procedure Call (RPC) programming model for the Cell Broadband Engine Architecture (CBEA). CBE RPC is a facility for calling a SPU procedure from a PPU program as if it were a regular PPU procedure call. To the application programmer, a SPU function call looks like a PPU call, and there are several components that work together to implement this facility, including the Interface Description Language, its compiler, and the IDL runtime library.

In the following section we provide the motivation for implementing a RPC-like programming model for the CBEA. In section 3, we describe the steps an application programmer needs to go through to develop a typical CBE RPC application. In section 4 we describe the CBE RPC internals. A brief description of the Interface Description Language is discussed in section 5. In section 6 we provide two examples. In the last section, conclusions are drawn and future work is considered. Two appendices on the user options for the IDL compiler and a complete IDL language synopsis are also provided.

2 Motivations

With a heterogeneous processing environment, the CBEA poses many programming challenges to the application programmer. This specific CBE RPC implementation is designed with the intention of enabling programmers to quickly write software for the Cell Broadband Engine. Using a familiar programming model, the CBE RPC implementation provides application programmers an environment to start developing applications for the architecture quickly and effectively.

The CBE RPC programming model is quite simple. Using a combination of runtime library and generated stub code, the PPU program can make calls to SPU functions seamlessly. Furthermore, each SPU function is loaded onto the SPU only once. The programmer can make multiple calls to the same function without having to load it again.

The asynchronous nature of the RPC programming model provides an easy way for programmers to exploit parallelism without having to understand the low level working of the MFC DMA layer.

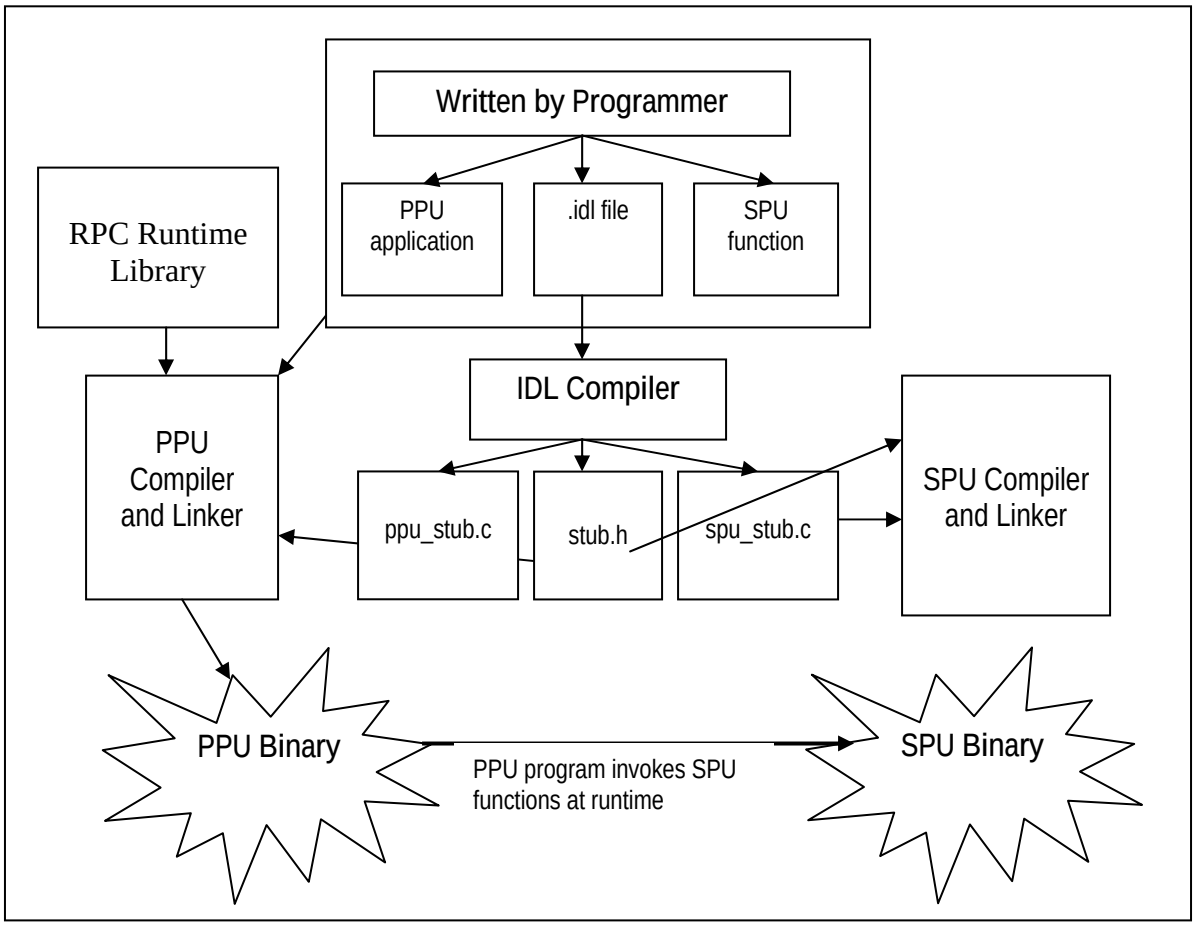
3 Overview of CBE RPC application development steps

To write a CBE program using this RPC implementation, the application programmer

needs to provide an interface definition file that defines the PPU/SPU interface. This interface, defined in the Interface Definition Language (IDL), consists of a set of prototypes for the SPU functions that the PPU will request the SPU to execute. The program is then compiled with the CBE IDL compiler. The output of the IDL compilation is a C header file and two C files, one for the PPU and one for the SPU. These two output C files contain the PPU and SPU stub code, where all the details of execution, data transfer, and so on, are managed in conjunction with the RPC runtime.

The generated header file is included in the two generated C files as well as PPU and SPU source files. It contains all the declarations and data structures that result from the interface definitions.

The IDL generated stubs make the PPU and SPU appear to communicate directly through ordinary, local procedure calls or method invocations. The stubs, together with the RPC runtime, enable the SPU to pull the necessary input parameters from the PPU to execute the procedure and push the output parameters to a pre-allocated place on the PPU once the SPU finishes executing the procedure. Figures 1 illustrates how the generated stub code works with the runtime library to enable SPU function invocation with parameters passing from PPU.



4 CBE RPC internals

4.1 Generated stub code

The IDL compiler uses lex and yacc to process the IDL file. The results of the compilation are one C header file and two C files. The C header file includes function declarations that are defined in the two C files. It also includes definitions for data structures that both PPU and SPU stubs need.

For each function 'foo' defined in the IDL file, the following data structures are defined:

`idl_foo_param_info`

This data structure keeps all the allocated pointers for input and output parameters of SPU function 'foo'

`idl_foo_work_item`

This data structure describes all the parameters and necessary information for the PPU and SPU to prepare parameters for marshaling and demarshaling to and from the SPU.

For each prototype function 'foo' defined in the IDL file, the following functions are defined:

`idl_id_t foo ()`

This is the stub code for function 'foo'. The PPU program can seamlessly invoke the SPU function 'foo' through this function. This function creates the **`param_info`** structure and sets all the appropriate parameters so the SPU stub code can DMA necessary parameters to local store.

Returns an integer identifying the function. This is useful for the PPU program to query the status of the SPU execution.

`int idl_join_foo (idl_id_t idl_func_id)`

Waits for the function execution with the idl_id_t func_id to finish executing. This function blocks until the SPU function finishes executing and sends a signal back to the PPU.

`int idl_poll_foo (idl_id_t idl_func_id)`

Polls the SPU execution of the function with idl_id_t func_id.
Returns 1 if the function execution has finish; 0 otherwise.

`void idl_join_all_foo()`

Waits for all the executions of function 'foo' to finish
Returns void.

For each interface 'foo_interface', the following functions are defined:

`void idl_foo_interface_cleanup()`

After finish calling all the functions defined in the .idl interface, an application must call this function to free up all the allocated resources

This function returns void

4.2 PPU runtime library

The RPC runtime library initializes the SPUs, loads the appropriate SPU functions to local store, and manages the task queue. PPU requests for SPU executions are represented as task structure. As each SPU function is invoked, a new task is created and put on a task queue. The tasks are scheduled according to a round-robin policy. There is a static number of slots on the queue. As soon as a SPU function returns, a slot on the queue is free up. If the queue is full, the PPU application will have to wait for a free slot in the queue.

4.3 How it all works together

The PPU program calls the function 'foo'. The stub function 'foo' initializes the SPU with necessary data and code, pack the parameters, and sends a mailbox message to the SPU to start the SPU stub program. The mailbox message contains the address of the work_item data structure which the SPU pulls over using MFC get commands.

5 Interface Description Language

The CBE RPC Interface Description Language is a subset of the DCE Interface Description Language. It also contains some new key words to allow CBE specific features like double buffering and vector type. A full grammar synopsis can be found in Appendix B.

5.1 Keywords and Reserved Words

IDL contains keywords and reserved words. These must not be used as identifiers. The following are IDL keywords and reserved words:

```
interface
int
char
double
float
vector
in
out
sync
async
size_is
dbuf_size
import
```

5.2 Identifiers

Each object is named with a unique identifier. The maximum length of an identifier is 31 characters. Some identifiers are used as a base from which the compiler constructs other identifiers. These identifiers have further restrictions on their length. The character set for identifiers is the alphabetic characters A to Z and a to z, the digits 0 to 9, and the _ (underbar) character. An identifier must start with an alphabetic character or the _ (underbar) character.

The CBE IDL is a case-sensitive language.

5.3 Comments

The **/*** (slash and asterisk) characters introduce a comment. The contents of a comment are examined only to find the ***/** (asterisk and slash) that terminate it. Thus, comments do not nest. One or more comments may occur before the first non-comment lexical element of the IDL source, between any two lexical elements of the IDL source, or after the last non-comment lexical element of the IDL source.

5.5 Interface Definition Structure

An interface written in IDL has the following structure:

```
<interface> ::=  <interface_header>  { <interface_body> }
```

5.6 Interface Header

```
<interface_header> ::= "interface" <identifier>
```

5.7 Interface Body

```
<interface_body> ::= [ <import> ] ... [ <export> ] ...
```

5.8 Import

```
<import> ::= “import” filename “;”
```

5.9 Export

```
<export> ::=  <const_declaration> ";"
            | <type_declaration> ";"
            | <op_declaration> ";"
```

5.10 Constant Declaration

```
<const_declaration>::=
"const" <const_type_spec> <identifier> "=" <const_exp>

<const_type_spec> ::=
int_type
| char_type
| string_type
| void_ptr_type
```

```
<const_exp> ::=
integer_const_exp
| character_const_exp
| string_const_exp
| identifier
| NULL
```

In the production <string_const_exp>, no character is permitted to be the doube quote character unless it is immediately preceded by the \ (backslash) character. In the production <character_const_exp>, no character maybe the single quote character unless it is immediately preceded by the backslash character.

LITERAL_INTEGER may appear only if <const_type_spec> is **long, short, int**
LITERAL_CHARACTER may appear only if <const_type_spec> is **char**
LITERAL_STRING may appear only if <const_type_spec> is **char***
NULL may appear only if <const_type_spec> is **void***
An <identifier> must have been defined in a preceding constant declaration. The type that <identifier> was defined to be in that declaration must be consistent with the <const_type_spec> in the current declaration.

In the generated code, constants are declared as **#define** macros

5. 11 Operation Declaration

```
<op_declaration> ::=
<op_attribute>  "idl_id_t" <identifier> <parameter_declarators>
```

The syntax for <op_attribute> is

```
<op_attribute> ::= ["sync" | "async" | "async_i" | "async_b"]
```

The keyword **sync** specifies synchronous execution. The PPU application must wait for the SPU to finish executing before continuing executing

The keyword **async_b** specifies asynchronous execution. The PPU application returns as soon as it copies the in parameters over. The user an reuse the in-buffer after the function returns. An idl_id_t is returned to the PPU. PPU can call **join_func (idl_id_t id)** function later to sync up the results.

The keyword **async_i** specifies asynchronous execution. The PPU application does not have to wait for the SPU to finish executing. The user cannot reuse the in-buffer after the function returns. An idl_id_t is returned to the PPU. The PPU program can call **join_func (idl_id_t id)** function later to sync up the results (see **IDL compiler user guide**) The user cannot reuse the in-buffer after the function return until after a successful join_func.

The attribute **async** has the same semantics as the attribute **async_i**

An operation must return a value of the opaque type **idl_id_t**. This serves as a handle in the asynchronous execution case; the programmer can use this **idl_id_t** handle to synchronize the results of the program.

The <identifier> in an operation declaration is the name of the operation (this is just the name of the function)

Each parameter declaration in an operation declaration takes the following form:

```
[<parameter_attributes>] <type_specifier> <parameter declarator>
```

Parameter declarators and attributes are declared separately in the following section.

5.12 Parameter Declarations

A parameter declaration takes the following form:

```
[<parameter_attributes>] <type_specifier> <parameter_declarator>
```

A parameter attribute can be any of the following:

in: the parameter is an input parameter
out: the parameter is an output parameter

size_is(val): the parameter has a size of 'val', 'val' can be a previously declared constant or parameter or any integer)

dbuf_size(val): if the specified parameter is an array, then this parameter is considered for double buffering. The parameter has a size of 'val', 'val' can be a previously declared constant or parameter or any integer. This value must be divisible by the array size. Furthermore, the double buffer size (in bytes) must be a multiple of 128. If these conditions are not met, the double buffering request is ignored.

The directional attribute **in** and **out** specify the directions in which a parameter is to be passed. The **in** attribute specifies that the parameter is passed from the PPU to the SPU. The **out** attribute specifies that the parameter is passes from the SPU to PPU

An output parameter must be passed by reference and therefore must be declared either as an array or a pointer. The size of the array and pointer must be given as **in** parameter prior to the declaration of the **out** parameter. Example:

```
[async] idl_id_t foo ([in] int size, [out, size_is(size)] int
ret_array[])
```

In this example, *size* is declared before *ret_array* and *size* is also specified as the size of the out parameter *ret_array*

5.13 User defined type

User-defined type right now is not implemented in the idl language. However, the programmer can import a header file to be included in all the generated source code files. This header file can contain any user defined type needed in the **.idl** file.

A pointer to a user defined structrure must have **size_is(int val)** as an attribute

Example: if the programmer wants to include the file "vse_subdiv.h" in all the generated stub code file. The **.idl** file will have this:

```
interface vse_subdiv
{
import "vse_subdiv.h";
[sync] idl_id_t foo ([in] int size,[in, size_is(size)] MyStruct* ptr);
}
```

The definition for *MyStruct* is in the file **"vse_subdiv.h"** and *size* is the size of the structure.

5.14 Arrays

IDL supports one dimensional array with the following features

Fixed: The size of the array is defined in IDL at compile time (this can be a declared constant or an integer in the **size_is(int val)** attribute)

Variable size: The size of the array is determined at runtime. The user can declare the size of the array by using the **size_is(int val)** attribute with **val** being an **in** parameter passed to the SPU at runtime.

An array parameter must have the **size_is** attribute specifying the number of the elements of the array, example:

```
[in, size_is (array_a_size)] int array_a
```

with array_a_size being an integer

6 Examples

We provide two examples of how to use the CBE RPC layer in an application. The first example is a simple 'hello_world' program. This is just a simple illustration of how everything fits together. In actuality, it is not recommended to have such a simple program running on the SPU. In the second program, a more complex example is given. Using the RPC facility, a PPU program invokes different functions from a SPU math library. All the SPU functions are executed asynchronously and only when the results are needed that the PPU program queries for SPU completion status.

6.1 Simple example: hello world

In this example, the PPU will invoke the SPU function `hello` from inside the PPU program execution. This simple program is an illustration of how all the components of the

RPC layer work together.

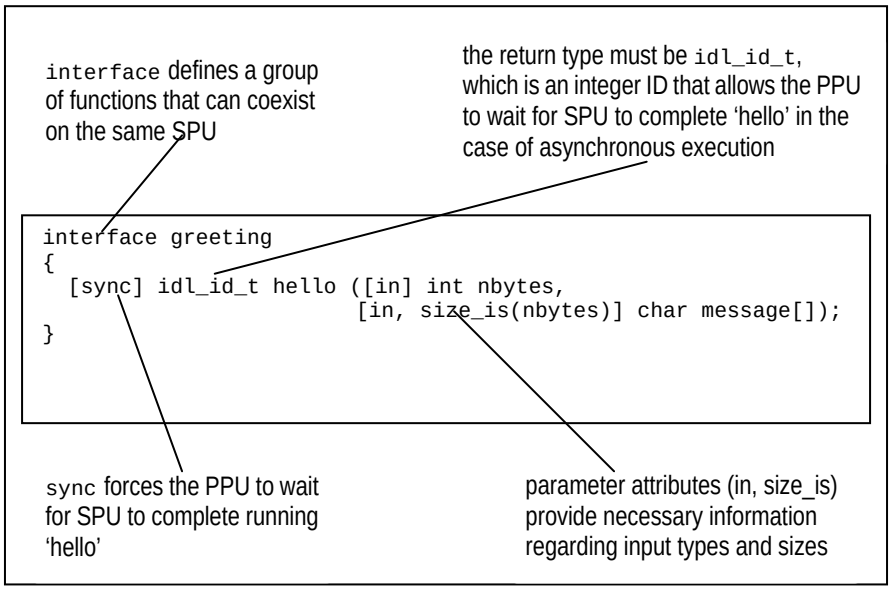


Figure 3 lists the content of ppu_hello.c

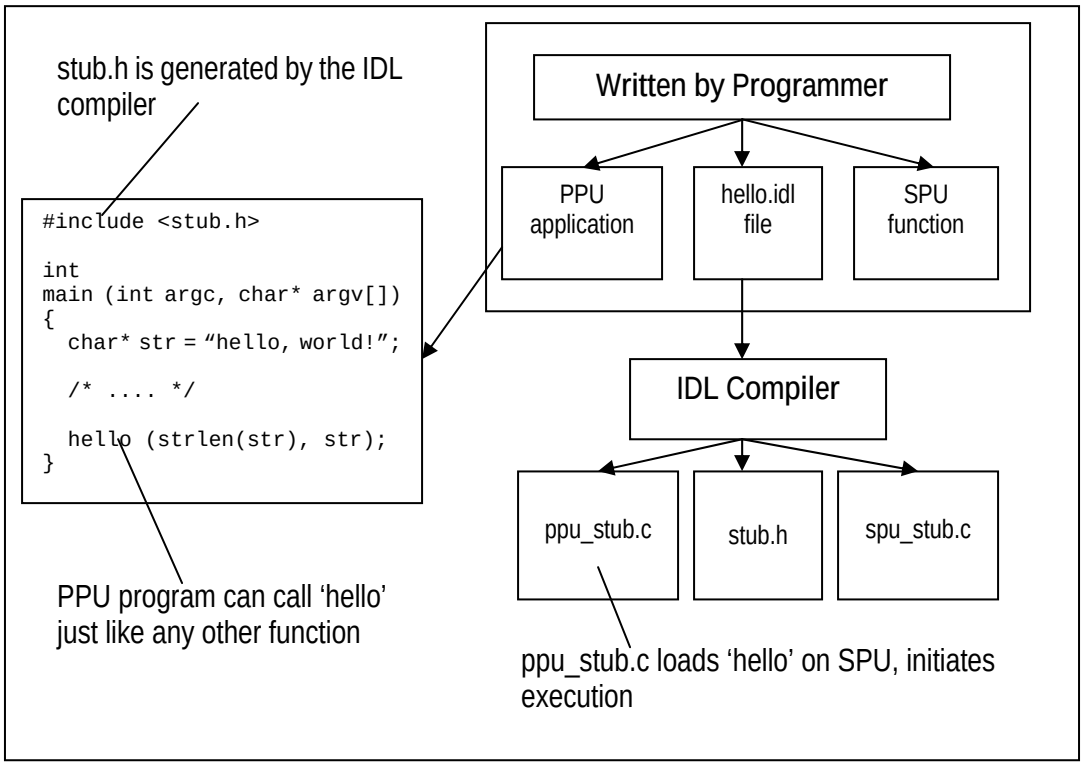
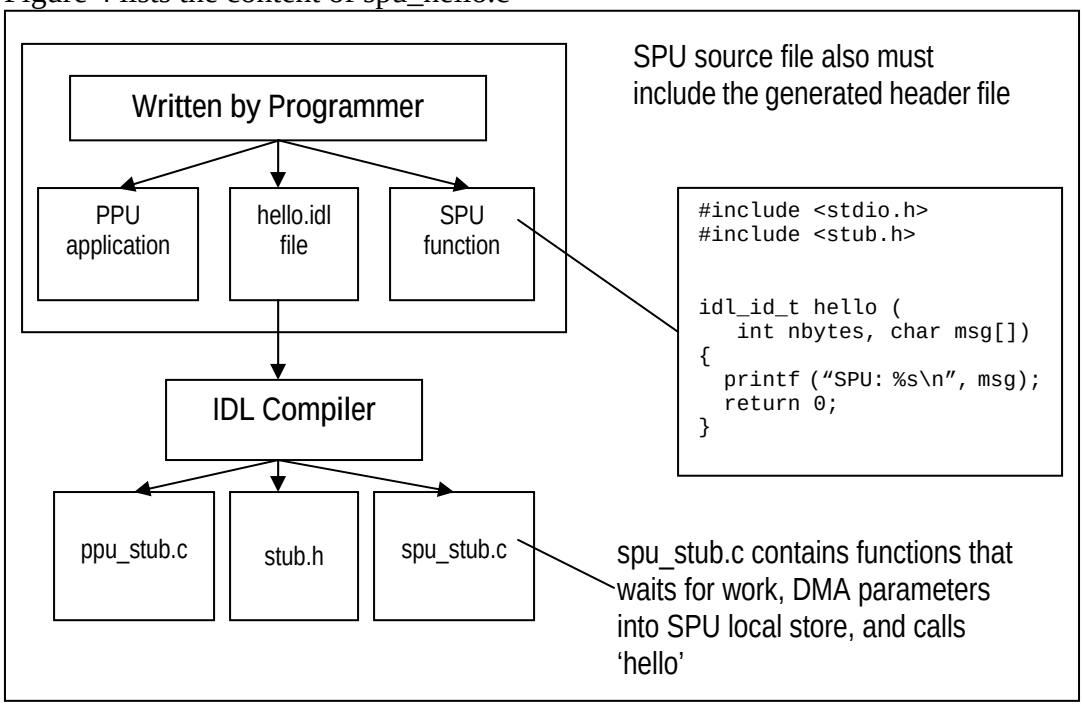
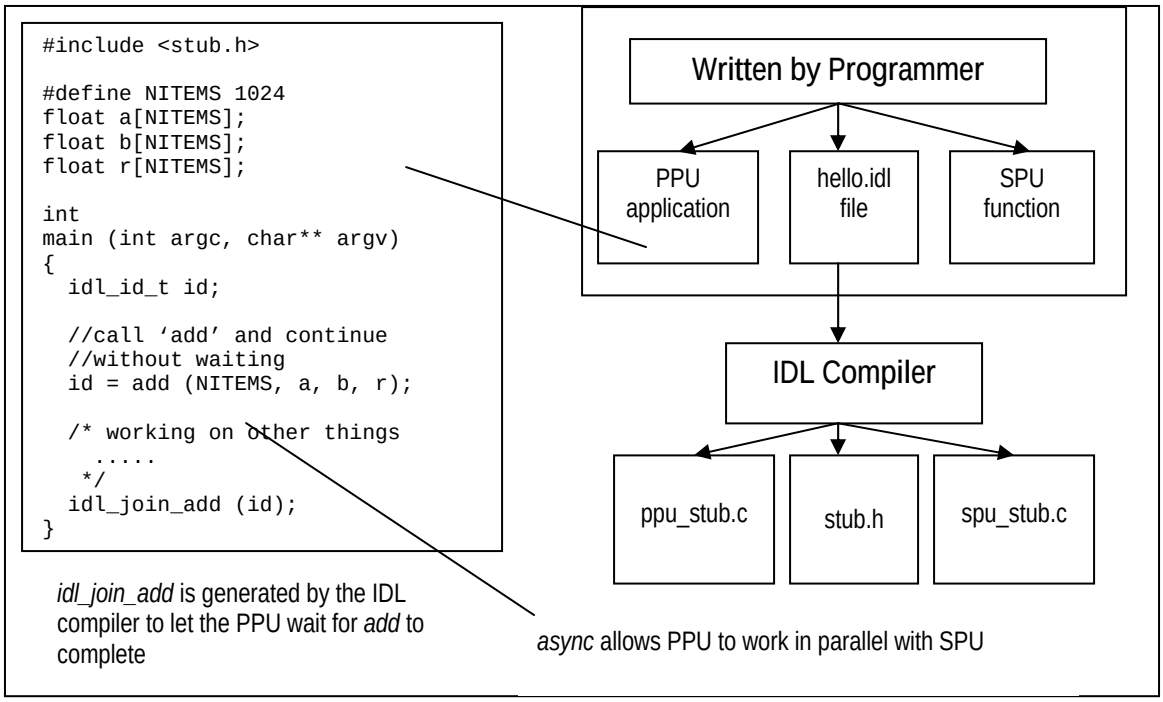


Figure 4 lists the content of spu_hello.c



6.2 Asynchronous example: math library

In this example, we have a math library composed of four functions, add, subtract, multiply, and divide. Each of these functions operates on array inputs that the SPU need to pull from system memory to SPU local store. The generated stub code of course handles all of the data transfer between PPU and SPU. The PPU can invokes these functions, let the SPUs execute them in parallel while doing other works, then when ready issue a join statement to wait for them to complete.



7 Conclusion and future works

The RPC programming model provides a familiar and easy environment for application programmer to start developing applications for the CBE architecture. The marshaling and de-marshaling of parameters to and from the SPU are hidden from the application layer.

The drawback to using the RPC layer is that it does not fit for performance critical application. The generated stubs and the IDL runtime impose certain performance penalty. Furthermore, the subset of the DCE Interface Description Language used in this implementation is rather limited. The application programmers will no doubt require more flexibility to express their SPU functions.

In the future, a more complete RPC solution could be implemented for the CBE architecture. The interface description language here is but a subset of the original DCE IDL that we are modeling after. A more robust double buffering scheme is also very beneficial to streaming applications.

8 References

1) IBM Distributed Computing Environment for AIX, Version 2.2: Application Development Guide

Appendix A - IDL compiler user options

```
The idl compiler is a single program (idl). It first parses an IDL input file
(.idl) and then produces C code for the stubs.

Usage: idl [<options>[ <idl-file>
Options Description
-h      shows this help

-c      if this option is set, the idl compiler will generate code to check
        whether there's enough memory in the local storage to bring the
        parameters over. An error message is returned to the programmer
        at runtime of the generated stub code if there's not enough memory
        in the SPU LS. If this option is not set, who knows what would happen?
        The stack may blow, the text section may be overwritten...
        No check is generated on default.

-d      sets output directory. The default is the current directory

-f      sets the name of the stub header file. The default is stub.h

-s      sets the name of the stub code for spu. The default is spu_stub.c

-p      sets the name of the stub code of ppu. The default is ppu_stub.c

-n      sets the maximum number of spus the ppu stub code can allocate.
        NOTE: spus are allocated on a need-based policy. If there's no need to
        allocate more spus (allocated spus are currently idle), extra spus
        will not be allocated,even if the maximum number of spus are not
        reached yet.

-b      the name of spu elf file the ppu will load. The default is the name of
        the interface in the .idl file

-e      If this flag is set, the idl compiler will output an echo.idl with the
        same content as the input idl file

-k      If this flag is set, the idl compiler generates code that's can be compiled
        as c++ code

-r      sets the name of the README file. The default README
        (not implemented yet)

-i      If this flag is set, all the parameters that can be double buffered
        will be pulled into local storage incrementally. For example, if you
        have an in parameter of 128K, but you only have 100K of local storage
        left, and this parameter can be double-buffered, then you would want
        to set this flag and let the SPU pulls only 64K of data in at a time.
```

```
        This option allows programmer to call function with large parameters
        that don't fit into SPU local storage

-g      If this flag is set, profiling information will be printed out. The
        following profiling information is provided:
            _ the number of bytes SPU DMAed in
            _ the number of bytes SPU DMAed out
            _ the number of cycles (amount of time) the SPU takes to DMA in
              parameters
            _ the number of cycles (amount of time) the SPU takes to DMA out
              parameters
            _ the number of cycles (amount of time) the SPU takes to execute
              the actual spu function
            _ the total number of cycles (amount of time) the SPU takes to run
        If anyone needs more information, please let me know
        For double buffering, the number of cycles the SPU spent to pull in
        parameters cannot be known exactly, so that information isn't
        reported.
        For synchronous execution, performance information is printed out
        after the function finishes executing.
        For asynchronous execution, performance information is printed out
        in the wait_spu_func function

Example: to generate stub files for the file sample.idl, the command would be:

    ./idl -p ppu/stub_sample.c -s spu/stub_sample.c -n 4 sample.idl
```

Appendix B – IDL Grammar Synopsis

<idl_interface> ::= <interface_header> "{" <interface_body> "}"
<interface_header> ::= "interface" IDENTIFIER

<interface_body> ::=
  <interface_body> <import>
 | <interface_body> <export>
 | empty

<import> ::= "import" <import_file> ";"
<import_file> ::= FILENAME

<export> ::=
  <const_declaration> ";"
 | <op_declaration> ";"

<const_declaration> ::= "const" <const_type_spec> IDENTIFIER "=" <const_exp>

<const_type_spec> ::=
  <int_type>
 | <float_type>
 | "char"
 | "char *"
 | "boolean"

<const_exp> ::=
  <integer_const_exp>
 | <float_const_exp>
 | <character_const_exp>
 | <string_const_exp>
 | IDENTIFIER
 | EXPNULL

<integer_const_exp> ::= LITERAL_INTEGER

<float_const_exp> ::= LITERAL_FLOAT

<character_const_exp> ::= LITERAL_CHAR

<string_const_exp> ::= LITERAL_STRING

<op_declaration> ::= <op_attribute> <idl_id_type> IDENTIFIER
<parameter_declarators>

<idl_id_type> ::= "idl_id_t"

```
<op_attribute> ::= "[" <op_attr> "]"

<op_attr> ::=
          "async"
        | "async_b"
        | "async_i"
        | "sync"

<parameter_declarators> ::=
          "(" "void" ")"
        | "(" <parm_decs> ")"

<parm_decs> ::=
          empty
        | <parm_dec_list>

<parm_dec_list> ::=
          <param_declarator>
        | <parm_dec_list> "," <param_declarator>

<param_declarator> ::=
        <param_attributes> <type_spec> <declarator>

<param_attributes> ::=
        "[" <parm_attrs> "]"

<parm_attrs> ::= <directional_attribute> <param_attribute>

<param_attribute> ::=
          empty
        | <field_attribute>

<directional_attribute> ::=
          "in"
        | "out"
        | "inout"

<field_attribute> ::=
        "size_is" "(" <attr_var> ")"

<attr_var> ::=
          LITERAL_INT
        | IDENTIFIER
<type_spec> ::=
          <simple_type_spec>
```

```
        | <constructed_type>

<constructed_type> ::= IDENTIFIER

<simple_type_spec> ::= <base_type_spec>

<base_type_spec> ::=
            <int_type>
          | <float_type>
          | <char_type>
          | <boolean_type>
          | <byte_type>
          | <void_type>
          | <vector_int_type>
          | <vector_float_type>
          | <vector_char_type>

<float_type> ::=
            "float"
          | "double"

<int_type> ::=
            "int"
          | <signed int>
          | <unsigned_int>

<signed_int> ::=
            "signed int"
          | <int_size> "int"
          | <int_size>

<unsigned_int> ::=
            "unsigned int"
          | <int_size> "unsigned int"
          | <int_size> "unsigned"
          | "unsgined" <int_size> "int"
          | "unsigned" <int_size>

<int_size> ::=
            "long"
          | "short"

<char_type> ::=
            "unsigned" "char"
          | "char"
```

```
<boolean_type> ::=
          "boolean"

<byte_type> ::=
         "byte"

<void_type> ::= "void"

<vector_char_type> ::=
         "vector" <char_type>

<vector_float_type> ::=
         "vector" <float_type>

<vector_int_type> ::=
         "vector" <int_type>
<declarator> ::=
           <complex_declarator>
         | IDENTIFIER

<complex_declarator> ::=
           <array_declarator>
         | "*" IDENTIFIER

<array_declarator> ::=
        IDENTIFIER "[" "]"
```