UNH C* Compiler Internals

May 24, 1994

This is an internals document for the UNH C* Compiler. Each section corresponds to a major functional unit. Unfortunately, this document only reflects recent work.

Contents

1 C* Parse Tree Tester

1.1 Introduction

Compilers normally produce a parse tree which is then turned into a simpler graph structure. The simpler structure is then run through various optimization phases and finally passed on to a code generator. The UNH C* compiler has the somewhat unusual requirement of being capable of producing C which strongly resembles the input C* code (especially when few C* specific constructs are used). This has the unfortunate side effect of requiring the initial parse tree to be comprehensive in its representation of C. Moreover, in order to generate output code which is similar in appearance to the input, final code generation is done from the parse tree and not a simpler intermediate form.

Since code generation is done directly on the parse tree, so too must all optimization transformations. The complexity of the parse tree data structure makes transformations of any sort fraught with danger. Since there is no guarantee that subsequent operations will detect an improperly modified parse tree, isolation of problems relating to improper transformations could be difficult.

The purpose of the parse tree tester is to implement a comprehensive sanity check on the parse tree. The normal use of this library is expected to be gross level fault screening and isolation between compiler phases which modify the parse tree and subsequent phases which use the parse tree. Testing on a more discrete level is possible however, if that is desired.

1.2 Using the Library

The library has two primary entry points, `GraphNodeTest()` and `GraphTreeTest()`, which allow for validity testing of a single parse tree graph node or an entire parse tree fragment respectively.

1.3 Implementation Issues

1.3.1 Source Files

`grphtest.c` - The complete implementation of the parse tree test library.

`grphtest.h` - External definitions for the library.

1.3.2 Design

While there are some common features of every node that are always tested (self referential pointer, node tag range validity, no more than a single parent) the procedure for testing any given type of graph node can be substantially different. Internally, the graph tester contains a table with entries corresponding to every graph node type. The table entry encodes such information as the number and types of child nodes which the node may have. Graph nodes which have complicated internal consistency requirements are accommodated by a table field which points at an optional node tag specific test function.

1.3.3 External Dependencies

The graph tree tester obtains all external graph related information from `graph.h` and `grphutil.h`. It uses the compiler standard message system (`message.h`) to display any faults.

2 Data Flow Graph Builder

2.1 Introduction

Since the UNH C* compiler was meant to be able to emit C code which strongly resembles the input C* code, the ordinary compiler step of turning the input parse information into intermediate code for various optimization transformations was neither necessary nor desirable. In UNH C*, code generation is done directly from the parse tree.

The nature of typical intermediate code is such that a data flow graph is essentially obvious based on the "go to" orientation of such code. Lacking such code, it was necessary to provide UNH C* with a substitute which not only provided a data flow graph, but did so in such a way that information generated from data flow analysis would be appropriately distributed in the original parse tree.

2.2 Using the Library

The data flow graph builder has a single entry point, "`Dfg_Build()`, which accepts the parse tree generated by the compiler front end. The builder defines a property type (managed by the property list system) for transition arcs. For any given node tagged with the property, the value is a list of other nodes to which control could flow. The caller may select among the following options for handling of expression trees:

- Do not draw arcs through expression trees, only connect expressions into the flow graph through their root node.
- Draw arcs through expression trees, assume a left to right evaluation order for operations which have no defined evaluation order.
- Draw arcs through expression trees, assume a right to left evaluation order for operation which have no defined evaluation order.

It may be worth observing that the ANSI C standard states that the handling of any expression which is evaluation order sensitive is undefined.

2.3 Implementation Issues

2.3.1 Source Files

`dfgbuild.c` - The complete implementation of the data flow graph builder.

`dfgbuild.h` - External definitions for the data flow graph builder.

2.3.2 Design

The data flow graph builder constructs the control flow graph for a function parse tree in a single traverse of the parse tree. The traverse mimics that which might be accomplished by an intermediate code generator and, instead of generating code, connects arcs from previously visited nodes to the current node being visited.

2.3.3 External Dependencies

The data flow graph builder has complete knowledge of the details of the graph node record structure, using definitions from `graph.h` and `grphutil.h`.

```
PL_HANDLE       Df_Duchain(
  void          *start,         /* Start node of the data flow graph */
  PL_HANDLE     df_arcs,        /* Data flow transition arcs (list of
                                void *) */
  PL_HANDLE     use_list,       /* Uses per node (list of (void *)) */
  PL_HANDLE     def_list,       /* Definitions per node (list of
                                (void *)) */
  void          *(*mem_alloc)(unsigned int len),
  void          (*mem_free)(void *mem)
                                /* Memory allocators */
);
```

Figure 1: Prototype for DU chain computation routine

3 Generic Data Flow Analysis Library

3.1 Introduction

The data flow analysis library provides a generic implementation of the algorithms which yield DU chains, UD chains, and loop extent information for an input data flow graph with variable use and definition information.

3.2 Using the Library

In order to remain independent of the internal compiler structures represented in a data flow graph node, the data flow analyzer accepts input and returns results in the form of property handles. The calling compiler is responsible for using the property list system to create a property environment for all the nodes which make up the data flow graph. It is then responsible for using the property list system to assign the various types of data expected by the data flow analyzer with appropriate nodes in the environment. See the property list library documentation for specific information on how property list data is managed.

The library has a number of external entry points which support operations with data flow graphs and incremental entry points to the loop identification and designation process. The key entry points are documented in following sections.

3.2.1 Computing DU Chains - `Df_Duchain()`

Figure 1 contains the prototype for this routine. This routine accepts three properties, representing data flow arcs, variable uses, and variable definitions associated with any given graph node.

Each property represents a value list. The data flow arcs are naturally simply `(void *)` items indicating any transitions from a particular node. The other two lists consist of `(void *)` pointers from the same set universe, indicating specific values which are used or defined at that node, respectively.

Internally, this function uses the input information to compute gen and kill sets and perform the normal computation of reaching definitions. Intermediate information is not retained after completion of the function.

The function result is a property assigned to all the use nodes specified by the use property parameter. This value property is a list of `Rd_Chain` structures specifying the originating node and defining value which could reach this node.

```
int             Df_Loops(
  void          *start,         /* Start node of data flow graph */
  PL_HANDLE     df_arcs,        /* Data flow arcs property */
  PL_HANDLE     *loopstack,     /* Pointer to graph property to fill
                                in with the loop stack graph property.
                                May be null if not required by caller. */
  PL_HANDLE     *loopdoms,      /* Pointer to graph property to fill
                                in with the loop dominator graph
                                property.  May be null if not required
                                by caller. */
  void          *(*mem_alloc)(unsigned int len),
  void          (*mem_free)(void *mem)
                                /* Memory allocators */
);
```

Figure 2: Prototype for loop extent computation routine

```
PL_HANDLE       Df_Loopinvar(
  PL_HANDLE     du_chain,       /* Property specifying DU chains */
  PL_HANDLE     lp_queue,       /* Property specifying loop queues */
  PL_HANDLE     lp_dominators   /* Property specifying the inner-most
                                loop dominator for all nodes which
                                are components of a loop */
);
```

Figure 3: Prototype for variable use loop invariance computation routine

3.2.2 Computing Loop Extents - `Df_Loops()`

Figure 2 contains the prototype for this routine, the purpose of which is simply to run the entire set of component functions (also available externally if lower level control is required) for computing loop extent information. This results in calls to `Df_Dominators()`, `Df_Backarcs()`, `Df_Naturalloops()`, `Df_Actualloops()`, and `Df_Loopqueues()`. A final call to `Df_Loopdoms()` is issued only if the `loopdoms` parameter is non-null.

The resulting properties provided by this routine allow for comprehensive understanding of the loop structure in effect at any point in the caller's data flow graph. Every node in the data flow graph, which participates in any loop, will have a defined value for the `loopdoms` property. The value of this property is the loop dominator for the inner-most loop which the node participates in. The `loopstack` property is only present in nodes which dominate loops. The value of the property is a stack which indicates the loop nesting structure in effect at that point in the data flow graph. The top element of the stack is the inner-most loop. Progressive elements of the stack are dominators for progressively outer loops.

While the caller need not accept a result for `loopstack`, that result is a necessary intermediate step for computing `loopdoms`.

3.2.3 Computing Variable Use Loop Invariance - `Df_Loopinvar()`

Figure 3 contains the prototype for this routine. This routine accepts the standard data flow information to determine the degree of loop invariance (that is, the outer-most loop) within which a given variable use is invariant.

3.3 Implementation Issues

3.3.1 Source Files

The generic data flow analyzer is made of the following source files:

`df.h` - External definitions for the analyzer.

`dfgutil.c` - Functions which support operations with data flow graphs.

`loopdoms.c` - Functions which determine the loop extents of a data flow graph.

`rchgdefs.c` - Functions which determine reaching definitions (DU and UD chains) for a data flow graph.

3.3.2 Design

The algorithms used to compute the data flow information are entirely conventional in nature, and this document will not attempt to reproduce what is best found in [1].

To the extent that this data flow implementation is unique, that uniqueness is the result of the use of the property list manager to manage input parameters, incremental results, and final results.

The property list system also allows for trivial sequential access to data with a given property or set of properties. While it is typical, not all of the data flow algorithms require the nodes of the graph to be visited in any particular order. In such cases, the implementation will normally create a function which defines the operations for any given node being processed by the algorithm. The implementation also can define a property which selects those nodes of the complete environment that are necessary. Both the function and the property defining which nodes to select are handed to the property list system, which then applies the function to all of the selected nodes in some arbitrary sequence. In many cases, this allows algorithms normally implemented using some form of depth first search to be implemented iteratively.

3.3.3 External Dependencies

The data flow analyzer was designed to be compiler independent. It has no knowledge of compiler specific internal information such as fields of a parse tree node or the like. The three libraries supporting the data flow analyzer are:

Property List Management Library - `plist` - Facilitates management of data which is spread about on an arbitrary set of nodes.

List Management Library - `lists` - A generic doubly-linked list manager with a comprehensive set of access functions.

Set Management Library - `sets` - A dense universe set manager implementing the usual logical set operations.

4 Restricted Communication Operations

The general approach behind all restricted communication operations is to optimize an iterated communication pattern by learning characteristics of the pattern on the first iteration and exploiting this knowledge on subsequent iterations. This opportunity for performance enhancement was first recognized by Joel Saltz, et. al. [5, 6] who produced primitives that could be explicitly used within the context of Fortran-D programs. Research at UNH has shown that this optimization strategy can be completely integrated within the context of the standard parallel language C*, without the need for any special efforts on the part of the application programmer.

It has been found that there are reasonable semantics for restricted variants of a number of communication types used by UNH C*, among these are:

- Scatter
- Gather
- Grid Write
- Grid Read

UNH C* also recognizes communication types known as broadcast, reduce, and owner assign. However, reasonable restricted variants of these have not yet been determined for the execution model assumed by the UNH C* compiler [4].

4.1 Compiler Operation

The compiler uses conventional data flow analysis to locate communication points which may be appropriately implemented in restricted form. Any of the recognized communication points is searched for, and the components of the communication are checked for loop invariance in the usual way [1].

Provided that some degree of loop invariance is determined, the compiler returns to an examination of the original program syntax to determine whether the C* context modifying statements `WITH`, `WHERE`, or `EVERYWHERE` may affect the active element context at the communication point under scrutiny. If the active element context also exhibits some degree of loop invariance for the candidate communication point, then the communication may be properly implemented in restricted form.

4.2 Modifications to the parse tree

Modifications to the parse tree necessary for emitting a restricted version of a communication operation, once properly identified, is straight forward. A tree node field, in the root node of the communication expression, is assigned a sequential value indicating which communication handle is associated with this communication point. In the root node of the entire parse tree, a total is incremented indicating that space for an additional communication handle will be required when the routine initializes at run–time. Finally, a node is inserted into the parse tree immediately above the loop node within which the restricted communication is known to be invariant. This node indicates the point at which the communications handle, associated with this restricted communication, should be marked as invalid (the initial condition).

5 Multi-Computer Environment Back-End

WARNING: *This section is adapted from a rather old technical report on C* run-time support for the Intel Delta. It has been somewhat updated, but is out of date and obsolete in many respects. It is retained here for now however, since it represents the only documentation of a number of critical features. For exact and up to date information on the interface to the run-time support, consult the* `CS__defs.h` *header file.*

```
struct CS__shape_struct
   {
   int      CS__num_local;          /* Number of local VPs                  */
   int      CS__max_local;          /* Max number of local VPs on any node */
   int      CS__num_total;          /* Total number positions in the shape */
   int      CS__min_vp_rows;        /* Min. vp rows per physical processor */
   int      CS__min_vp_cols;        /* Min. vp cols per physical processor */
   int      CS__max_vp_rows;
   int      CS__max_vp_cols;
   int      CS__local_rows;         /* Number of rows local to this node   */
   int      CS__local_cols;         /* Number of cols local to this node   */
   int      CS__rows_above;         /* No. rows held on nodes above this   */
   int      CS__cols_above;         /* No. cols held on nodes left of this */
   int      CS__row_extra;          /* # row pos. that don't divide evenly */
   int      CS__col_extra;          /* # col pos. that don't divide evenly */
   int      CS__row_break;          /* Offset of first vp after break point*/
   int      CS__col_break;          /* Offset of first vp after break point*/
   char    *CS__context;            /* Pointer to current context          */
   int      CS__rank;               /* Rank of shape (-1 if unspec)        */
   int      CS__dims[CS__MAX_RANK]; /* Length of each axis                 */
   int      CS__prods[2][2][CS__MAX_RANK]; /* Partial products -- all nodes */
   int      CS__prods_local[CS__MAX_RANK]; /* Partial products -- local     */
   CS__Map  CS__map[CS__MAX_RANK];   /* Mapping for each axis               */
   };

typedef struct CS__shape_struct *CS__Shape;
```

Figure 4: C structure that encapsulates all the information for a shape.

5.1 Run-time Representation of Shape

5.1.1 Overview

Each variable of the new C* type `shape` is represented in the emitted C code as a pointer to a structure, with fields as described in Figure 4.

All of the structure fields shown in this figure are initialized when the C* shape variable becomes fully specified. See Figure 5 for a sample shape declaration and the C code which is emitted for it.

Note that the shape variable `s` is represented in the C code as a pointer to a compiler-generated structure, `CS__temp_0`, which is initialized by `CS__allocate_shape_array_1()`. This function is capable of initializing an entire array of shapes, as in this example declaration: `shape [5]sarr[10]`.

```
/* C* shape declaration */
shape [10]s;

/****************************************************************/

/* Emitted C code */
struct CS__shape_struct CS__temp_0;
CS__Shape s = &CS__temp_0;

/*
**  This function is executed before main().
*/

void
_GLOBAL__I_SHAPE_test_cs_0(void)
{
  CS__SetUp();
  CS__allocate_shape_array_1 ((CS__Shape *) &s, (CS__Shape) &CS__temp_0,
           sizeof(s) / sizeof(CS__Shape), 1, (int) (10));
}
```

Figure 5: Sample C* shape declaration and generated C code.

The compiler emits functions whose names begin with `_GLOBAL__I_`. These functions are called "start-up functions" and are executed before user's main function begins. This is the method used by the GNU Project's `g++` compiler to call constructors and destructors of global objects.

We have added some script-support for systems which do not support this GNU feature. This script examines object files, collects the names of all startup functions, and emits a small C file that defines a function called `CS__InitShapesAndPvars` that in turn invokes each startup function. The UNH C* compiler arranges to have the generated function invoked before any user-supplied code is executed[1], producing the illusion that shapes were somehow initialized at compile time. It is impossible to actually initialize shapes at compile time because the size of the mesh on which the program will be executed is unknown to the compiler.

Note that every startup function begins with a call to `CS__SetUp()`. This allows the run-time library to perform any internal allocation and initialization. The second and subsequent calls to `CS__SetUp()` are ignored by the library.

5.2 Run-time Representation of Parallel Variables

This section will cover in some detail how parallel variables are represented in the C output of the UNH C* compiler. The small sample program shown in Figure 6 will be used as an example throughout this section.

Note that the C output for line 1 of the sample program was described in the previous section,

[1]This is done by generating the user's main program as `CS__UserMain` and then generating a main program that first calls `CS__InitShapesAndPvars` and then calls `CS__UserMain`.

```
1  shape [10]s;
2
3  int:s  intvar;
4  int:s *ptrvar, *ptrvar2;
5  int:s  arrvar[5];
6
7  struct x { double mem1; int mem2; } strvar:s;
8
9  int i;
10
11 main()
12 {
13    with (s)
14       {
15       intvar      = 1;
16       strvar.mem1 = 2;
17       strvar.mem2 = 3;
18       arrvar[0]   = 4;
19
20       ptrvar = &intvar;
21       ptrvar = arrvar;
22       ptrvar = &arrvar[1];
23       ptrvar = &strvar.mem2;
24
25       ptrvar = ptrvar + 1;
26
27       i = ptrvar - ptrvar2;
28       }
29 }
```

Figure 6: Sample C* program for this section.

and will not be covered again here.

5.2.1 Parallel Base Types

This section describes the simplest type of parallel variables: those with an arithmetic base type.

Representation

Line 3 of Figure 6 declares `intvar` to be a parallel integer of shape `s`. As with all parallel variables, `intvar` is represented in the emitted C code as a `CS__Pvar`, which is actually a `char *` and points to the start of the local portion of the data associated with the parallel variable. However, the storage for each parallel variable is prepended with a header, `CS__PvarHeader`. See Figure 7.

The `CS__PvarHeader` structure contains a pointer to the shape associated with the parallel variable, a pointer to the beginning of the data associated with the parallel variable, and the stride (the distance in bytes between each virtual processor's data) of the parallel variable.

Lastly, the field `CS__align` is used to guarantee proper alignment of VP data. (We assume that `long double` has the strictest alignment requirement of any base type.) When the parallel variable is allocated, a `CS__PvarHeader` is "over-allocated" so that one call to the memory allocation function will return enough space for both this structure header and all of the local VPs' data. (The

```
typedef struct {
   CS__Shape      CS__shape;   /* Ptr to associated shape                  */
   char          *CS__base;    /* Ptr to beginning of data space           */
   int            CS__stride;  /* Bytes between subsequent elements        */
   long double    CS__align;   /* Ensure correct alignment for all types */
   }  CS__PvarHeader;
```

Figure 7: The C structure used to represent parallel variables.

```
CS__Pvar intvar;

/*
**  This function is executed before main().
*/

void
_GLOBAL__I_VAR_test_cs_1(void)
{
  CS__SetUp();
  intvar = CS__PvarAlloc ((s), sizeof(int), CS__USER_MEM_POOL);
}
```

Figure 8: C output for `int:s intvar` declaration.

actual allocation is done by the run-time library.) Then `CS__base` is made to point at the address of `CS__align`, which guarantees that `CS__base` points at enough properly aligned memory to hold all local VP values.

The data to which `CS__base` points is allocated so that each VP's data is located in contiguous memory. That is, all of the data associated with any single virtual processor will be located in contiguous memory locations. This only becomes important when discussing parallel arrays and structures, but also holds true for the simple parallel integer case described here.

Note that the `CS__base` field contains no notion of the actual type of data associated with the parallel variable. This field is always a vanilla `char *`. It is the compiler's job to emit the proper typecasts to ensure correct behavior.

Figure 8 shows the C code emitted for line 3 of Figure 6. `CS__PvarAlloc()` takes the shape, the stride, and an indication of where to allocate memory. The value of the `CS__base` field is what is returned by the `CS__PvarAlloc` function. If access to the header info is required later, then negative offsets can be applied to the pointer value[2]. Note that the `CS__base` field value is not strictly required to be kept both in the header and as the value of the parallel variable itself. The redundant copy is kept in the header for consistency checks.

Data Access

Now that `intvar` has been allocated, how do we store values into each VP's position? Figure 9 shows the C output for line 15 of our sample program.

[2]The macros `CS__get_stride` and `CS__get_shape` in `CS__defs.h` support access to the header.

```
int CS__vpi = 0;     /* VP index for emulation loops */
int CS__limit = 0;   /* VP index limit */

register char *CS__temp_5 = (intvar);      /* base of data */
register int   CS__temp_6 = (sizeof(int)); /* stride */

/* VP emulation loop starts here */
for (CS__limit = CS__current->CS__num_local,
       CS__context_ptr = CS__current->CS__context,
       CS__vpi = -1;
    ++CS__vpi < CS__limit;
    /* Nothing */)
    {
    if (*CS__context_ptr++)
      {
        (* (int *) CS__temp_5) = 1;
      }

    CS__temp_5 += CS__temp_6;     /* Get to next VP's data */
    }
```

Figure 9: C output for intvar = 1 assignment statement.

The declarations of CS__vpi and CS__limit are emitted at the beginning of every function that will require them. (Normal C functions are passed through the C* compiler without change.) They are used to represent the index of the current local virtual processor being operated upon, and the total number of local VPs on this node. They are assigned values at the beginning of the virtual processor emulation loop. (CS__current is a run-time object that always points to the most recently selected shape. In this case, it points to the C structure pointed to by s, as seen in the with statement on line 13 of the sample program.)

The declarations of CS__temp_5 and CS__temp_6 represent, respectively, the location of the current VP's data and the stride to get to the next VP's data. Note that the compiler is aware of the type of the parallel variable and can set the stride directly. The CS__stride field of the CS__PvarHeader is only required for more complex cases, such as pointers to parallel variables, which are discussed in later subsections.

The figure next shows the actual VP emulation loop, which processes each virtual processor's data one vp at a time. The testing of the value pointed to by CS__context_ptr is done to ensure that only active positions are evaluated. See the discussion of the where and everywhere statements in [3].

You can see that on each iteration of the VP emulation loop, CS__temp_5 points to the next local position of intvar to be assigned the value 1. This pointer is cast to the appropriate type (in this case, a pointer to integer) and the assignment is done. Then, CS__temp_5 is incremented by the stride contained in CS__temp_6, and the next pass of the VP emulation loop begins.

```
/* Declaration (int:s *ptrvar) */
CS__PvarPtr ptrvar;

/* compiler generated temporary */
CS__PvarPtr CS__ptemp_1;

/* Assignment (ptrvar = &intvar) */
ptrvar = (CS__ptemp_1.start=intvar,
          CS__ptemp_1.base=intvar,
          CS__ptemp_1);
```

Figure 10: C output for pointers to parallel variables.

```
CS__PvarPtr CS__ptemp_2;

ptrvar = (CS__ptemp_2.start=strvar + (CS__offsetof (struct x , mem2)),
          CS__ptemp_2.base=strvar,
          CS__ptemp_2);
```

Figure 11: C output for ptrvar = &strvar.mem2 assignment.

5.2.2 Pointers To Parallel Variables

A pointer to a parallel variable, as ptrvar on line 4 of Figure 6, is represented in the emitted C code as a CS__PvarPtr structure. This structure contains two fields, which are both of type char *. The base field points to the base of the parallel variable into which the pointer to parallel variable is pointing. From the base field the parallel variable header may be accessed. The start field points to the start of the data for this pointer to parallel variable.

Figure 10 shows the code emitted for the declaration of ptrvar on line 4 and for the assignment ptrvar = &intvar on line 20 of the sample program.

Note that the base and start fields might not be the same. Consider line 23 of the sample program. Figure 11 contains the C code generated for line 23. A parallel value is referenced that is a "slice" of the parallel struct. The base field of the CS__PvarPtr structure still points to the base of the encompassing parallel variable, so that the header can be referenced in the usual way with a negative offset. (Note that the stride is still the same: the distance between two elements is still the length of the struct.) The start field however points to the location of the first mem2 member. The CS__offsetof macro from CS__defs.h (identical to the ANSI C offsetof) returns the necessary adjustment to the base value.

5.2.3 Arrays of Parallel Variables

We allocate arrays of parallel base types as if the array itself were parallel. That is, only one CS__Pvar structure is allocated for the entire array, and each virtual processor allocates enough space to hold its entire array as if the array were a singular arithmetic base type. The stride for such a "parallel array" is the number of bytes to get from one VP's array element to the corresponding array

```
CS__Pvar arrvar;

/*
**  This function is executed before main().
*/

void
_GLOBAL__I_VAR_test_cs_2(void)
{
  CS__SetUp();
  arrvar = CS__PvarAlloc ((s), sizeof(int [5]), CS__USER_MEM_POOL);
}
```

Figure 12: C output for int:s arrvar[5] declaration.

```
void:void *user_alloc (shape s, int size)
{
   return palloc (s, size);
}

/* Later, in user code: */
pointer1 = user_alloc (s, boolsizeof(int:s));
pointer2 = user_alloc (s, N * boolsizeof(int:s));
```

Figure 13: Pathological palloc() array example.

element of the next VP.

The C code emitted for the allocation of arrvar is shown in Figure 12. Note that the stride associated with arrvar is simply the size of one VP's local array.

Allocating arrays of parallel variables within one run-time parallel variable data structure is dictated by the malloc-like semantics of the palloc C* library routine. The palloc routine does not know if it is allocating space for an array or not, and therefore must handle both cases in the same manner. See Figure 13 for an example.

```
CS__PvarPtr CS__ptemp_3;

ptrvar = (CS__ptemp_3.start=arrvar,
          CS__ptemp_3.base=arrvar,
          CS__ptemp_3);
```

Figure 14: C output for ptrvar = arrvar assignment.

```
CS__PvarPtr CS__ptemp_2;

ptrvar = (CS__ptemp_3.start=arrvar + (1)* sizeof(int ),
          CS__ptemp_3.base=arrvar,
          CS__ptemp_3);
```

Figure 15: C output for ptrvar = &arrvar[1] assignment.

5.2.4 Parallel Array/Pointer Conversions

In C, an array name used in a value context decays into a pointer to the first element of the array. In our implementation of C*, this decay of array to pointer requires that we emit some simple conversion code.

Take line 21 of Figure 6, for example. Here, arrvar is used in a value context, and is being assigned to ptrvar. Because we effectively push "arrayness" into the C run-time structure but keep "pointerness" outside the structure, the code in Figure 14 is emitted by the compiler. Note that neither the start address nor the stride of the parallel variable changes during this conversion.

Figure 15 shows a similar conversion for the assignment on line 22. Now, because ptrvar must be made to point at the second integer of each VP's array (element one), the start address of the pointer must be adjusted by adding 1 * sizeof(int).

Because the distance from one VP's array base element to the next VP's array base element has not changed, the stride of the new pointer variable is the same as the stride of the array. In fact, once a parallel variable is created, its stride never changes.

```
CS__PvarPtr CS__ptemp_5;

ptrvar =  (CS__ptemp_5.start=ptrvar.start + 1 * sizeof(int ),
           CS__ptemp_5.base=(ptrvar).base,
           CS__ptemp_5);
```

Figure 16: C output for ptrvar = ptrpvar + 1 assignment.

```
i = ((ptrvar.start - ptrvar2.start) / sizeof(int ));
```

Figure 17: C output for i = ptrvar - ptrvar2 assignment.

5.2.5 Parallel Pointer Arithmetic

Because of the way we represent pointers to parallel variables, arithmetic operations performed on them need minor editing to maintain the usual C semantics. We will examine the statements on lines 25 and 27 of the sample program in Figure 6.

Line 25, ptrvar = ptrvar + 1 increments a pointer to a parallel variable. (It is assumed that ptrvar enters this expression pointing to any but the last array element in a parallel array.) We emit the C code shown in Figure 16. This adjusts the start pointer so that it points one integer position past where it currently points.

Similar care must be taken when subtracting two pointers to parallel variables. As in C, pointer subtraction is only defined when both pointers point to different elements of the same array, and the result is the integral number of elements between the pointers. In our case, we must scale the pointer subtraction to account for the fact that each VP's data is contiguous. See Figure 17.

Note that the expression `ptrvar.start` and `ptrvar2.start` retrieve the current data pointers for the objects pointed to by `ptrvar` and `ptrvar2`. The pointer subtraction is then done on the `char *` pointers, and the scaling is accomplished by dividing by `sizeof(int)`, which is the type of the objects pointed at.

5.3 Communication Functions

In the following sections, we describe the UNH C* multicomputer back-end run-time interface specification. Techniques for implementation of these routines are found in various run-time internals documents.

5.3.1 C* Example

The Figure 18 contains sample C* declarations that will be used to illustrate the various examples throughout this section. Refer to [2] for a complete description of the language.

5.3.2 Broadcast

A *broadcast* is used to distribute a single value residing on one processor to all other processors in the topology. An example of C* code that causes a call to broadcast to be emitted by the compiler is the following:

```
scalar_f = [5]pv1[4].mem2;
```

This example shows a broadcast operation, or more specifically an *owner broadcast*. The value of position five of the fourth array element of `pv1` is assigned to a scalar, and conceptually the change must be distributed to all the copies of `scalar` across the topology so that they remain up-to-date. The value of `[5]pv1[4].mem2` is broadcast into a compiler-generated scalar temporary variable, and then every node does the assignment to `scalar_f` locally.

Figure 19 shows a fragment of the C code emitted by the C* compiler for this assignment, which conforms to the prototype for `CS_Broadcast` in Figure 20.

`dest` is a pointer to the scalar temporary generated by the compiler. `src` is the parallel variable from which the source value must be extracted by the library. In this example, `pos_size` is the size of operands of type "type" and `lindex` specifies the parallel left index. `lval_size` is the size for operands of type "type".

5.3.3 Reduce

Reduce is called when each active position of a parallel variable must contribute its value to some scalar expression. The code

```
scalar_f += pv1[4].mem2;
```

combines the sum of the values of all active positions of `pv1[4].mem2` with `scalar_f`. The reduction requires communication among all nodes for two purposes: each active position must contribute its value to the operation and all nodes must receive the new value for `scalar_f` following the reduction.

The prototype for this function appears in Figure 21.

```
#define MM    128
#define NN    128

struct struct1 {int mem1; float mem2;};
shape [MM] s1;                    /* Shape of rank one and dimension MM   */

struct struct1:s1 pv1[NN];       /* Declare an array of parallel structs */
float:s1          pv2, pv3, pv4; /* Float parallel variables             */
float             scalar_f;      /* Scalar float                         */

main()
{
   with (s1)       /* Initialization */
      {
      int i;

      /* Initialize the array of parallel structures */
      for (i = 0; i < NN; i++)
         {
         pv1[i].mem1 = 1;
         pv1[i].mem2 = 2.0;
         }

      pv2 = 3.0;
      pv3 = 4.0;
      }
}
```

Figure 18: C* sample declarations.

5.3.4 Scatter and Gather

Scatter and *gather* are called when parallel values are to be distributed in an unpredictable pattern across the nodes. The following code demonstrates the C* code requiring calls to scatter and gather:

```
[pv2]pv3 = pv4;   /* Scatter */
pv4 = [pv2]pv3;   /* Gather  */
```

5.4 Memory Pool Management

The compiler must emit C code that allocates memory for the storage of parallel variables and context masks. It uses a memory pool manager to perform these tasks. This section describes, in general terms, the role the memory pool manager plays in the execution of the translated C code.

5.4.1 Motivation

Consider the C* code fragment in Figure 23. This example shows three user-defined parallel variables (<u>`temp`</u>, <u>`temp2`</u> and <u>`temp3`</u>) that will be allocated with a different shape each pass through the <u>`for`</u> loop.

```
        float CS__temp_1;

        CS__lindex[0] = (unsigned long) (5);
        CS__lval_size[0] = sizeof (float );
        CS__src[0].start= pv1 + (4)* sizeof(struct struct1 ) +
                               (CS__offsetof (struct struct1 , mem2));
        CS__src[0].base= pv1;
        CS__pos_size[0] = sizeof (struct struct1);

        /* Do the communication */

        CS__Broadcast (
  0U, NULL, (void *) &CS__temp_1, CS__src, CS__FLOAT,
  (size_t) sizeof(struct struct1), 1, CS__lindex
);
```

Figure 19: C code generated for the call to CS__Broadcast.

If we chose an expensive method of allocating and deallocating parallel variables, this code would run very slowly. Further, performing one deallocation for each object would add to execution time and compiler complexity.

The solution we eventually adopted is a memory pool. For simplicity, assume that the memory pool is implemented as a stack. Our implementation allows the compiler to "mark" the current location in this stack (that is, the address the next allocation will begin at), allocate whatever space it needs, and quickly "reclaim" back to any mark.

In the above example, the compiler would mark the memory pool before performing the allocations, make the requests for space to allocate the parallel variables, and eventual perform one reclaim to push the memory pool back to the state it was in before the allocations.

The underlying memory allocated by the memory pool itself is never released—the pool never shrinks. This makes further allocations extremely fast if the memory pool has already grown large enough to accommodate the request.

5.4.2 Memory Pool Management Strategies

There are actually five separate memory pools maintained by the compiler. Here is a brief description of each, along with the mark/reclaim strategy used for each pool.

- CS__COMPILER_TEMP_MEM_POOL

  This pool is used when allocating any compiler-generated temporary variable. These allocations occur when a temporary is need to capture the result of a communication.

  Any single C* statement may have multiple communications. The compiler "lifts" these communications into a group that will be executed prior to the statement itself. The statement may then examine the compiler-generated temporary variables to retrieve the results of the communications.

  Because the results of any communication group are consumed by the immediately following statement, this memory pool can be reclaimed at the beginning of the communication group.

```
void*
CS__NBroadcast (
  unsigned int  modifiers,      /* Standard parameter, general purpose
                                communication modifiers                 */
  void          *context,       /* Standard parameter, context handle   */
  void          *dest,          /* Scalar dest of bcast                 */
  CS__PvarPtr   src,            /* Source pvar                          */
  CS__Type      type,           /* Type of the operands                 */
  CS__size_t    size,           /* Size of position                     */
  unsigned int  rank,           /* Rank of comm                         */
  unsigned int  lindex[]        /* Per axis left index ("rank" defines
number of elements in this vector)      */
)
```

Figure 20: The CS__Broadcast prototype.

```
void *
CS__NReduce (
  unsigned int  modifiers,      /* Standard parameter, general purpose
                                communication modifiers                 */
  void          *context,       /* Standard parameter, context handle   */
  void          *dest,          /* Scalar dest of reduction             */
  void          *src,           /* Source per-processor parallel temp   */
  CS__Type      type,           /* Position type of source              */
  CS__size_t    size,           /* Operand size                         */
  CS__RedOp     op              /* Reduction operator                   */
);
```

Figure 21: The CS__Reduce prototype.

The values associated with any previous communication group must have been used before the current group was executed.

The pool is marked on entry to each C* function, and is either reclaimed as described above, or when the function executes a `return` statement or falls off the end of the function.

- CS__USER_MEM_POOL

  This pool is used to allocate space for user-declared parallel variables. This pool is marked on entry to a block containing user parallel variable declarations, and is reclaimed when that block is exited (via `return`, `break`, `continue` or falling through the end of the block).

  Each function statically reserves enough space to store as many memory pool marks as it will need. This is determined by the maximum block nesting in the function, which is usually a very small number.

- CS__CONTEXT_MEM_POOL

```
void *
CS__Scatter (
  unsigned int  modifiers,      /* Standard parameter, general purpose
                                communication modifiers                  */
  void          *handle,        /* Standard communication handle parm.  */
  CS__PvarPtr   dest,           /* Dest. pvar                            */
  CS__PvarPtr   src,            /* Source pvar                           */
  unsigned int  rank,           /* Rank of comm. (passed to low level
                                scatter routines, but unused there).    */
  CS__PvarPtr   lindex[],       /* Per axis left index pvar              */
  CS__Type      dtype,          /* Type of destination operand           */
  CS__Type      stype,          /* Type of source operand                */
  CS__size_t    size,           /* Size of operand                       */
  CS__RedOp     op              /* Operation to perform                  */
);

void *
CS__Gather (
  unsigned int  modifiers,      /* Standard parameter, general purpose
                                communication modifiers                  */
  void          *handle,        /* Standard communication handle parm.  */
  CS__PvarPtr   dest,           /* Dest. pvar                            */
  CS__PvarPtr   src,            /* Source pvar                           */
  CS__Type      type,           /* Position type                         */
  CS__size_t    size,           /* Position size                         */
  unsigned int  rank,           /* Rank of communication                 */
  CS__PvarPtr   lindex[]        /* Per axis left index pvar (length of
                                this defined by "rank").                 */
);
```

Figure 22: The CS__Scatter and CS__Gather prototypes.

The context masks associated with shapes are allocated from this memory pool. This pool is marked when entering a where or everywhere statement, and is reclaimed when exiting that block.

As with CS__USER_MEM_POOL, each function statically reserves enough space to store the maximum number of marks the function will require. This number is the where (and everywhere) nesting level of the function.

- CS__PFUNC_RETURN_MEM_POOL

  This pool allocates space for functions which return parallel values. That is, the function will allocate space out of this pool to store the parallel return value.

  This pool is very much like CS__COMPILER_TEMP_MEM_POOL, except that the pool is not reclaimed when the function is exited. This ensures that the calling function will receive a valid parallel return value that has not been overwritten by further memory pool activity.

```
for (i = 0; i < MAX; i++)
   {
   int:(shape_func()) temp, temp2, temp3;

   /* Etc. */
   }
```

Figure 23: Memory pool manager motivation.

- CS__STATIC_PVAR_MEM_POOL

  This memory pool is used to declare user-defined parallel variables that have `static` storage duration. Essentially, each static parallel variable is allocated once, on the first entry to the block containing the declaration.

  Because static parallel variables must exist for the full duration of program execution, this pool is never marked, and reclaims are never done.

5.5 Profiling

The library provides the compiler with profiling capabilities, as well as having the ability to perform profiling on itself. While the compiler emits calls to begin and end profiling at a high level, the library is equipped to provide detailed information as to where the time is being spent in the library itself. What follows is a description of these profiling capabilities from both the compiler and library viewpoints.

5.5.1 Profiling Data Structure

```
typedef struct CS__p
   {
   unsigned long  CS__pro_count;  /* number of times "activated."   */
   int            CS__pro_type;   /* the structure's "type"         */
   char          *CS__pro_name;   /* the structure's "name"         */
   char          *CS__pro_file;   /* C* file assoc. with the struct */
   int            CS__pro_line;   /* C* line number assoc. w/struct */
   CS__ProType    CS__pro_acc;    /* timing "accumulator"           */
   CS__ProType    CS__pro_local;  /* time spent sorting/local work  */
   CS__ProType    CS__pro_send;   /* time spent in MsgWrite         */
   CS__ProType    CS__pro_rcv;    /* time spent in MsgRead          */
   CS__ProType    CS__pro_wait;   /* time spent waiting for msg     */
   } CS__profile;
```

Figure 24: The profiling data structure.

The run-time library has the ability to provide timing data for a single UNIX workstation and the Intel Touchstone Delta. CS__ProType is defined as a typedef according to the selected architecture's clock units. The fields in the structure will be described in further detail in the following sections.

5.5.2 Library Calls

For self-timing, the library is primarily concerned with four fields in the profiling data structure shown in Figure 24: CS__pro_local, CS__pro_send, CS__pro_rcv and CS__pro_wait. The first, CS__pro_local, is used to accumulate the time it takes for each communication function to perform all associated local work. CS__pro_send indicates the time required to perform the send operation in the form of the function call CS__MsgWrite, and similarly CS__pro_rcv provides the time spent in CS__MsgRead performing local work. Finally, CS__pro_wait contains the time spent waiting for messages to arrive while no other work is going on. The compiler provides the library with a pointer to the correct data structure in which to accumulate timings. If the pointer is non-null, the library assumes that the profiling system is activated.

As the last step in the process, the library is responsible for outputting the accumulated data structures in a reasonable fashion. We have found there to be a measurable cost to profiling, which is evident in the resulting timing figures. Any discrepancies between the recorded total time and the sum of the parts can be attributed to these profiling costs.

5.5.3 Compiler-generated C Code

Figure 25 shows a simple example of the C code emitted by the compiler to profile a communication point. Note that the declaration of CS__temp_45 initializes the CS__pro_count, CS__pro_type, CS__pro_name, CS__pro_file and CS__pro_line fields with values appropriate for this communication point. The remaining fields will be initialized to zeros of the appropriate type by the back-end C compiler and linker.

```
static CS__profile CS__temp_45 = { 0, 'C',
                "Binary-Reduce", "test.cs", 177
                /* Other fields zero'd by type */ };

/* Begin profiling */
CS__ProfileStart (&CS__temp_45);

/* Do the communication (code omitted) */

/* End profiling */
CS__ProfileStop (&CS__temp_45);
```

Figure 25: Sample C profiling code.

5.5.4 Restrictions

Because we do not have access to the startup or cleanup functions performed by the underlying C environment, there are two restrictions that must be followed when profiling C* programs.

First, if any profiling is to be performed, the file containing main() must be compiled for profiling. Other files may be selectively compiled with or without the profiling flag.

Second, the C* program must exit by "falling through" the closing brace of main(). Any calls to exit(), abort(), etc. will prohibit the output of the collected profiling information.

6 Supporting Tools

The following section documents tools which support the implementation of the UNH C* Compiler, though none of them are specific to compiler applications.

6.1 Property List Manager Library

6.1.1 Introduction

Property lists are a type of list abstraction that allows the arbitrary association of data with a large set of related data elements. Collectively, the set of data elements are referred to as the environment, while individual members are nodes. Specific types of information which are to be attached to one or more nodes are known as properties and are referred to by property handles.

6.1.2 Using the Library

The first step in using the property list library, is to define the environment. This is done by creating a set of functions which allow the property system to peruse the members of the environment, and to access a `(void *)` pointer unique to each node.

A property handle is defined to return only one of the following two types of data:

value - A `(void *)` pointer value.

flag - Simple TRUE if present.

Property handles themselves are of two different types, depending upon how the property data is retrieved for a node.

actual - Environment nodes contain explicit value for the property. User calls establish the property value(s) at specific nodes.

interpreted - When the property is established, the caller defines an interpreting function. Each time the property is queried for a given node, the original caller function is called with the specific node as a parameter. No data is explicitly tagged to nodes. The result of the interpreting function is the value of the property. Interpreted flag properties are similar in that they do not actually tag environment nodes with property information; however they are constructed differently. Instead of a single interpreting function, they represent a logical expression of component properties. Refer to the discussion of `Pl_Freeproperty()` for more information.

Whether a property is defined as interpreted or actual, the retrieval functions are identical.

A special word of caution is in order with regard to function parameters. A significant number of functions in the property management library accept caller specified functions as parameters. While the interface to these functions is well specified by the function prototypes, the caller obviously has substantial opportunity for creativity in the operation of these functions. Nevertheless, the functions should be pure (no global side effects) since the caller has no control over when or how often these functions are called.

6.1.3 Implementation Issues

Source Files

`plist.c` - The complete implementation of the property list manager library.

`plist.h` - External definitions for the library.

Design Property lists are entirely dynamic in nature. The only static data in the system is a pointer to the error handler. Multiple environments may be managed simultaneously. The data structure representing all environment data consists of three types of records as follows:

Property Handles - The user's view of the system is of the pointers to property handles. There may be several property handles that directly or indirectly refer to the same property data. With the exception of interpreted flag properties, all property handles refer to a property descriptor which is the actual description of the data associated with the property.

Property Descriptors - Actual property information, user functions on how to test and release property data, etc., are stored in a property descriptor. Property descriptors persist so long as one or more property handles refer to them.

Property Items - For non–interpreted properties, a physical list is constructed for each node of the environment. The elements of this list are the property items. The items refer back to the property descriptor which owns them, and they also contain the user's pointer for value properties.

External Dependencies The only significant external dependency is the lists library.

6.2 List Library

6.2.1 Introduction

The list library provides a comprehensive implementation of operations for various types of lists up through doubly linked lists. Included are FIFOs, stacks, arbitrary priority queues, and other variants.

6.2.2 Using the Library

The list library does not understand any particular type of list element. Instead, it manages elements which are `(void *)` pointers to elements of some type which the caller understands. Individual lists are characterized by a list handle which is intended as a blind data structure containing list administration data. All operations on any list require the handle as a parameter.

6.2.3 Implementation Issues

Source Files

`lists.c` - The complete implementation of the list library.

`lists.h` - External definitions for the list library.

Design Regardless of how they are created, all lists are internally represented as doubly linked in order to maintain constant time performance for most typical list operations. The list system maintains no static data. All operations either create list handles or are performed as indicated by administrative data in the list handle structure. A smaller supporting structure is used to contain the `(void *)` pointer which characterizes each list element.

External Dependencies The lists library has no significant external dependencies.

6.3 Set Operations Library

6.3.1 Introduction

The set operations library provides an implementation of the usual set operations for set environments small enough to enumerate every instance of the set in a bit vector.

6.3.2 Using the Library

The set operations library relies upon the caller to maintain the actual representation of the set environment. The caller is responsible for keeping track of how many elements are in the set environment, and which element corresponds to each offset ($0..n-1$, where n is the number of elements in the environment).

6.3.3 Implementation Issues

Source Files

`sets.c` - The complete implementation of the sets library.

`sets.h` - External definitions for the sets library.

Design Individual set instances are represented as appropriate length bit vectors. The length of the set is rounded up to the next even `int` boundary. Set operations are thus performed on, presumably, elements which are the most convenient size for the underlying architecture.

External Dependencies The sets library has no significant external dependencies.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] J. Frankel. C* language reference manual. Thinking Machines Corporation, 1991.

[3] K. Herold. A retargetable C* communication and run-time library for mesh-connected MIMD multicomputers. Master's thesis, University of New Hampshire, 1992.

[4] M. Quinn and P. Hatcher. Compiling SIMD programs for MIMD architectures. In *IEEE International Conference on Computer Languages*, pages 291–296, March 1990.

[5] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. Technical Report 90–59, NASA Langley Research Center, NASA Langley Research Center, 1990.

[6] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. Technical Report 91–13, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1991.