High-Level Information – An Approach for Integrating Front-End and Back-End Compilers

Sangyeun Cho, Jenn-Yuan Tsai[†], Yonghong Song[‡], Bixia Zheng, Stephen J. Schwinn[*], Xin Wang, Qing Zhao, Zhiyuan Li[‡], David J. Lilja[*], and Pen-Chung Yew

Dept. of Comp. Sci. and Eng., Univ. of Minnesota, Minneapolis, MN 55455

[†]Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL 61801

[‡]Dept. of Comp. Sci, Purdue University, West Lafayette, IN 47907

[*]Dept. of Elec. and Comp. Eng., Univ. of Minnesota, Minneapolis, MN 55455

http://www.cs.umn.edu/Research/Agassiz

Abstract

Existing compilers for automatically parallelizing application programs are often divided into parallelizing front-end compilers and optimizing back-end compilers. The front-end compilers perform high-level program analysis to identify dependences among relatively coarse-grained program units, such as subroutines and loop iterations, and perform transformations on these large units. The back-end compilers, on the other hand, perform analyses and optimizations on relatively finer-grained operations, such as machine instructions. Due to a lack of high-level information, however, the back-end compilers miss potential optimizations, and the scope of the optimizations is limited to only short-range local transformations. In this paper we propose a new universal High-Level Information (HLI) format to effectively integrate front-end and back-end compilers by passing front-end information to the back-end compiler. Importing this information into an existing back-end leverages the state-of-the-art analysis and transformation capabilities of existing front-end compilers to allow the back-end greater optimization potential than it has when relying on only locally-extracted information. A version of the HLI has been implemented in the SUIF parallelizing compiler and the GCC back-end compiler. Experimental results with the SPEC benchmarks show that HLI can provide GCC with substantially more accurate data dependence information than it can obtain on its own. Our results show that the number of dependence edges in GCC can be reduced by an average of 48% for the integer benchmark programs and an average of 54% for the floating-point benchmark programs studied, which provides greater flexibility to GCC's code scheduling pass. Even with the scheduling optimization limited to basic blocks, the use of HLI produces moderate speedups compared to using only GCC's dependence tests, when the optimized programs are executed on MIPS R4600 and R10000 processors.

Keywords: data dependence, data flow analysis, program optimization, parallelizing compiler, information compression

[0]A short version of this paper appears in the 1998 International Conference on Parallel Processing, Minneapolis, MN, August 1998.

1 Introduction

Existing compilers for automatically parallelizing application programs are often divided into two relatively independent components. The *parallelizing front-end* of the compiler is often a source-to-source translator that typically is responsible for performing loop-level parallelization optimizations using either automatic program analysis techniques, or user-inserted compiler directives. This front-end then relies on an *optimizing back-end* compiler to perform the machine-specific instruction-level optimizations, such as instruction scheduling and register allocation.

This front-end/back-end separation is common due to the complexity of integrating the two, and because of the different types of data structures needed in both components. For example, the front-end performs high-level program analysis to identify dependences among relatively coarse-grained program units, such as subroutines and loop iterations, and performs transformations on these larger units. Because it operates at this coarse level of parallelism granularity, the front-end does not need detailed machine information. Not incorporating machine-specific information into the front-end eliminates the time and memory space overhead required to maintain this extensive information. The back-end, however, needs machine details to perform its finer-grained optimizations.

The penalty for this split, though, is that the back-end misses potential optimization opportunities due to its lack of high-level information. For example, optimizations involving loops, such as scalar promotion or array privatization [11], are difficult to perform in the back-end since complex loop structures can be difficult to identify with the limited information available in the back-end, especially for nested loops. Furthermore, the scope of the optimizations the back-end can perform, such as improving instruction issuing rates through architecture-aware code scheduling [7, 10, 12, 17, 20], is limited to only short-range, local transformations. Another consequence of this split is that it is not uncommon for transformations performed in the front-end to be ignored, or even undone, in the back-end.

The types of high-level information that could be used by the back-end to enhance its optimization capabilities include the details of loop-carried data dependences, information about the aliasing of variable names, the results of interprocedural analysis, and knowledge about high-level, coarse-grained parallelization transformations. While this information could be easily passed from the front-end to the back-end in a completely integrated compiler, it is unfortunately very difficult to build an entire compiler from scratch. To be competitive, the new compiler must incorporate the state-of-the-art in both the front-end and the back-end, which is a massive undertaking.

Instead of building a completely new compiler, we note that the necessary information can be readily extracted from an existing parallelizing front-end compiler and passed in a condensed form to an existing optimizing back-end compiler. In this paper, we propose a new universal *High-Level Information* (HLI) format to pass front-end information to the back-end compiler. Importing this information into an existing back-end leverages the state-of-the-art analysis and transformation capabilities of existing front-end compilers to allow the back-end greater optimization potential than it has when relying on only locally-extracted information.

In the remainder of the paper, Section 2 presents the formal definition of the HLI format, showing what information is extracted from the front-end and how it is condensed to be passed to the back-end. Section 3 then describes our implementation of this HLI into the SUIF front-end parallelizing compiler [29] and the GCC back-end optimizing compiler [25]. Experiments with the SPEC benchmark programs [26] and a GNU utility are presented in Section 4 showing how the use of the high-level information provided by SUIF improves the dependence information available to GCC. Related work is discussed in Section 5, with our results and conclusions summarized in Section 6.

2 High-Level Information Definition

A High-Level Information (HLI) file for a program includes information that is important for back-end optimizations, but is only available or computable in the front-end [27]. As shown in Figure 1, an HLI file contains a number of HLI entries. Each HLI entry corresponds to a program unit in the source file and carries the high-level information corresponding to that program unit. Each HLI entry contains two major tables – a *line table* and a *region table*. The format and contents of these tables are described below.

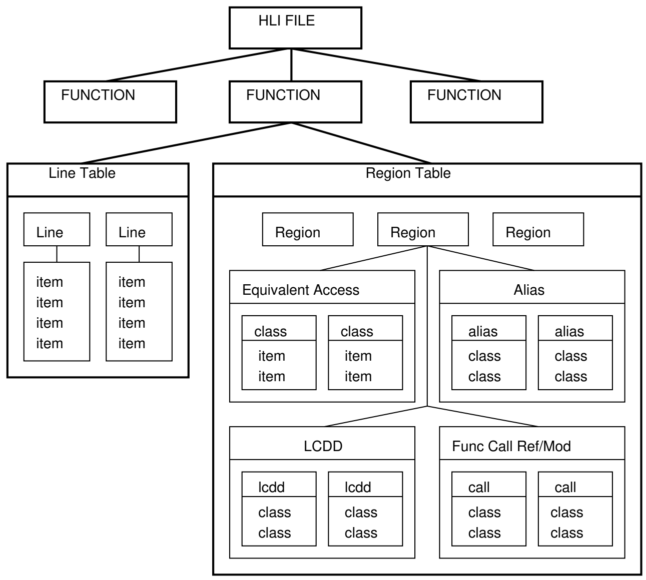

Figure 1: Top level layout of an HLI file.

2.1 Line table

The purpose of the line table is to build a connection between the front-end and the back-end representations. After generating the intermediate representation (IR), such as expression trees or instructions from the source program, a compiler usually annotates the IR with the corresponding source line numbers. If both the front-end and the back-end read the program from the same source file, the source line numbers can be used to match the expression trees in the front-end with the instructions in the back-end.

To reduce the amount of information that must be passed from the front-end to the back-end, the HLI focuses on only certain operations, such as memory accesses and function calls. These operations are called *items* in the HLI representation.[1] In the line table, the front-end lists the items associated with each source line. The front-end also assigns each item a unique identification number (ID), which is used by the region table to address these items.

A line table consists of a number of line entries where each entry corresponds to a source line of the program unit in the source file. Each line entry includes the source line number, the line type, the enclosed parent line number, and a list of item entries associated with the line. The line type indicates what kind of construct statement this line is, such as IF, FOR, ASSIGNMENT, etc. The parent line number points to the nearest enclosing control

[1]An item may also represent an equivalent access class or a whole region, as discussed in Section 2.2.

statement of this line. With both the parent line number and the line type associated with each program line, the back-end can reconstruct the high-level program structure to incorporate this information into the back-end representation.

In the item list, each item entry consists of an *ID* field and a *type* field. The ID field stores a unique number within the scope of the program unit that is used to reference the item. The type field stores the access type, such as load, store, function call, etc., of the item.

Groups of items from the front-end are mapped to the back-end instructions by matching their source line numbers. However, this mapping information may not be precise enough to map items inside a group (i.e. a single source line) from the front-end to the back-end. To perform precise mapping, the front-end needs to know the instruction generation rules of the back-end and the order of items associated with each source line. Specifically, the order of items listed in the line table must match the order of the items appearing in the instruction list in the back-end.

2.2 Region table

To simplify the representation of the high-level information while maintaining precise data dependence information for each loop, we represent the high-level information of a program unit with scopes of *regions*. A region can be a program unit or a loop and can include sub-regions. The basic idea of using region scopes in the HLI is to partition all of the memory access items in a region into *equivalent access classes* and then describe data dependences and alias relationships among those equivalent access classes with respect to the region.

The region table of a program unit stores the high-level information for every region in the program unit. Each region entry has a region header describing the ID, type, and scope of the region. In addition to the region header, each region entry holds four sub-tables: (1) an equivalent access table, (2) an alias table, (3) a loop-carried data dependence table, and (4) a function call REF/MOD table. In the following subsections, we describe each of these tables associated with each region.

2.2.1 Equivalent access table

A region can contain a large number of memory access items. Recording all of the data dependences and alias relationships between every pair of memory access items would result in a huge amount of data. In fact, many memory access items in a region may refer to the same memory location since the same variable may be referenced multiple times in a region. Such memory access items can thus be grouped into a single equivalent access class. Using equivalent access classes to describe the data dependences among memory access items greatly reduces the amount of high-level information that must be passed to the back-end.

The equivalent access table of a region partitions all memory access items inside the region, including those memory access items enclosed by its sub-regions, into equivalent access classes. The region table includes a number of different equivalent access classes. Each equivalent access class has a unique item ID, which can be used to represent all of the memory access items belonging to the class. The members of an equivalent access class can be either memory access items immediately enclosed by the region that are not enclosed by any sub-region, or the equivalent access classes of its immediate sub-regions. Equivalent access classes of immediate sub-regions are used to represent the memory access items that are enclosed by the sub-regions. In other words, memory access items which are not immediately enclosed by the region will not appear directly in the equivalent access table of the region. Instead, they are represented by the corresponding equivalent access classes at the sub-regions which enclose them. The equivalent access classes defined in a region must be mutually exclusive so that every memory access item inside the region, including those enclosed by its sub-regions, is represented by

```
int a[10];
int b[10];
int sum;

foo ()  /* region 1*/
{
    int i, j;

    for (i = 0; i < 10; i++) /* region 2 */
    {
        a[i] = 0;
        {1} /* item 1 */
        b[i] = i;
        {2}
    }

    sum = 0;
     {3}
    for (i = 0; i <10; i++) /* region 3 */
    {
        a[i] = a[i] + b[0];
         {4}    {5}    {6}
        for (j = 1; j  < 10; j++) /* region 4 */
        {
            b[j] = b[j] + b[j−1];
            {7}    {8}     {9}
            a[i] = a[i] + b[j];
            {10}   {11}   {12}
        }
        sum = sum + a[i];
         {13}   {14}   {15}
    }
}
```

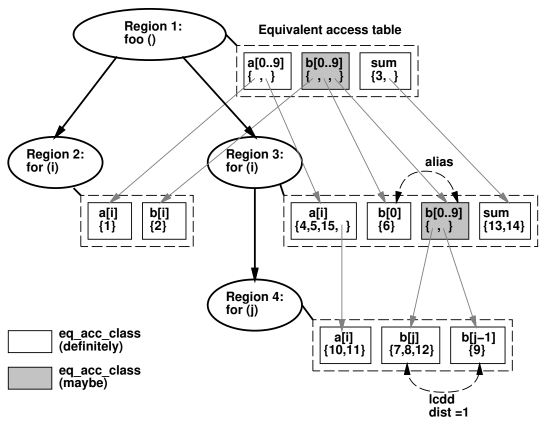

Figure 2: The structure of the regions and equivalent access classes for an example program.

exactly one equivalent access class in the region. However, a memory access item can be represented by different equivalent access classes defined at regions of different levels.

Typically, the memory access items of an equivalent access class are considered to be definitely equivalent since they always access the same memory location. However, the front-end compiler might want to group memory access items from different equivalent access classes at sub-regions that may access the same memory location into a single equivalent access class to reduce the amount of high-level information that must be passed to the back-end. In this case, the memory access items of an equivalent access class may not always access the same memory location. To distinguish this case, every equivalent access class has an *equivalent access type* field, whose value can be *definitely equivalent* or *maybe equivalent*. The property of maybe equivalence will propagate along the corresponding equivalent access classes in enclosing regions.

If a region is a loop, its equivalent access table only describes the equivalent access relationships among memory access items or sub-region equivalent access classes WITHIN a single loop iteration. Also, when referred to by the alias table and by the loop-carried data dependence table of the region, an equivalent class represents only the memory locations accessed in one loop iteration. However, when referred to by the outer region, the equivalent class represents all of the memory locations that will be accessed by the whole loop.

Figure 2 demonstrates the region structure of a procedure and its corresponding equivalent access tables. The outermost region of the procedure is Region 1, which represents the whole procedure. Region 1 has two immediate sub-regions (Regions 2 and 3) that represent the two *i* loops in the procedure. The second *i* loop (Region 3) has an inner *j* loop, which is represented by Region 4. In the equivalent access table of Region 1, all memory access items in the procedure are partitioned into three equivalent access classes: *sum*, *a[0..9]*, and *b[0..9]*. From the point of view of Region 1, every memory access item inside the procedure is represented by exactly one of those equivalent access classes. For example, in Region 1 Item 11 (*a[i]*) inside the *j* loop is represented by the equivalent access class of *a[0..9]*. As mentioned above, equivalent access classes use the IDs of sub-regions' equivalent access classes to refer to memory access items residing in their sub-regions. For example, the equivalent access class of *sum* in Region 1 uses the equivalent access class of *sum* defined in Region 3 to refer to memory access items 13 and 14 enclosed by Region 3.

2.2.2 Alias table

The alias table describes the possible alias relationships among the equivalent access classes of a region. Two or more equivalent access classes are said to be *aliased* if they may access the same memory location at run time. If two equivalent access classes are aliased, all of the memory access items represented by the two equivalent access classes are also aliased. Each alias entry in the alias table consists of a set of equivalent access classes that the front-end has determined to be aliased. The equivalent access classes in the alias table must be equivalent access classes that are defined at the current region. Since the alias table only describes the alias relationships among the equivalent access classes within a loop iteration, data dependences caused by equivalent access classes in different loop iterations will be described in the loop-carried data dependence table, described below.

While the equivalent access table of a region requires every memory access item inside the region to belong to exactly one equivalent access class at the region, the alias table allows an equivalent access class to appear in multiple alias entries at the region. Note that the alias relationship is not transitive. For example, if A and B are aliased and B and C are aliased, it is not necessarily implied that A and C are aliased.

In Figure 2, equivalent access classes *b[0]* and *b[0..9]* in Region 3 may access the same memory location. Thus the alias table of Region 3 will include an entry indicating that these two equivalent access classes are aliased.

2.2.3 Loop-carried data dependence table

If the region is identified as a loop, the *loop-carried data dependence table* will list all of the loop-carried data dependences caused by the loop. Loop-carried data dependences are represented by pairs of equivalent access classes defined at the region. Each pair specifies a data dependence arc caused by the loop. The data dependence type can be *definitely dependent* or *maybe dependent*. In addition, each dependence pair includes a distance field. To simplify the representation of the dependence distance, the direction of a dependence is always normalized to be '>' (forward), that is, from an earlier iteration to a later iteration.

For the example shown in Figure 2, the only loop-carried data dependence is between equivalent access classes *b[j]* and *b[j-1]* in Region 4. The distance of the loop-carried data dependence is one.

2.2.4 Function call REF/MOD table

The *function call REF/MOD table* of a region describes the side effects caused by function calls on the equivalent access classes of the region. If a function call is immediately enclosed by the region, the function call REF/MOD table will use the function call item ID defined in the line table to refer to the function call and will list the equivalent access classes that may be referenced or modified by the called function. For function calls inside

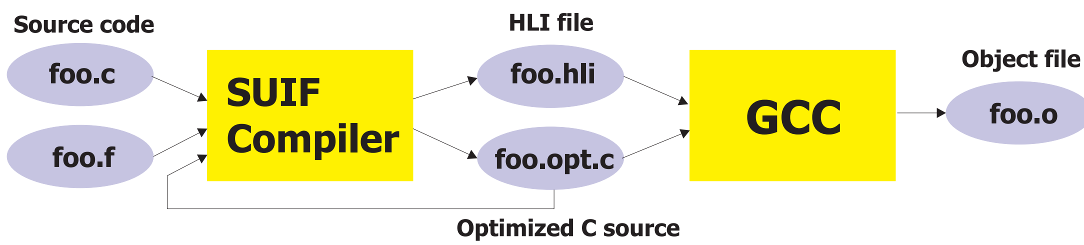

Figure 3: Overview of the HLI implementation using the SUIF front-end and GCC back-end compilers.

a sub-region, the function call REF/MOD table will use the sub-region ID to represent all of the function calls and will list the equivalent access classes that may be referenced or modified by the function calls inside the the sub-region. With the function call REF/MOD table, the front-end can pass interprocedural data-flow information to the back-end to enable the back-end to move instructions around a function call, for instance.

3 Implementation Issues

A version of the HLI described in the previous section has been implemented in the SUIF parallelizing compiler [29] and the GCC back-end compiler [25]. This section discusses some of the implementation details. Note, however, that the HLI format is platform-independent, and many of the implemented functions are portable to other compilers [24]. Figure 3 shows an overview of our HLI implementation in the SUIF compiler and GCC.

3.1 Front-end implementation

The HLI generation in the front-end contains two major phases – *memory access item generation* (ITEMGEN) and *HLI table construction* (TBLCONST). The ITEMGEN phase generates memory access items for each source line, and assigns a unique identification number (ID) to each memory access item. The memory access items for a source line, if ordered by the ID, can be one-to-one matched to the memory reference instructions in the GCC RTL chain[2] for the same line. These memory access items are annotated in the SUIF expression nodes to be passed to the TBLCONST phase.

The TBLCONST phase first collects the memory access item information from the SUIF annotation to produce the line table for each program unit. Next, it generates information for the equivalent access table, alias table, and loop-carried data dependence table for the program unit read and for each loop. Because the ITEMGEN phase is both back-end compiler dependent and machine dependent, while the TBLCONST phase is not, separating the HLI generation into these two phases allows us to reuse the TBLCONST programs when we use a different back-end compiler or target machine.

3.1.1 Memory access item generation

The ITEMGEN phase traverses the SUIF internal representation (IR) generating memory access items. It passes this memory access item information to the TBLCONST phase by annotating the SUIF IR. To guarantee that the mapping between the generated memory access items and the GCC RTL instructions is correct, we observe the following RTL generation rules:

[2] RTL (Register Transfer Language) is an intermediate representation used by GCC that resembles Lisp lists [25]. An RTL chain is the linked list of low-level instructions in the RTL format.

1. For a given assignment statement, first generate the memory accesses for the address subexpression on the left hand side. Then generate the memory accesses for the right hand side expression tree. Finally, generate the memory accesses to store the result of the right hand side to the destination address.
2. For any expression tree, first generate the memory accesses for the address subexpression of each operand. Then generate the memory accesses for operands that are also expressions. Finally, generate the memory accesses for operands that are variables.

Most of the memory access items correspond to variable accesses in the source program. However, when the optimization level is above -O0, GCC assigns a pseudo-register for a local scalar variable. An access to this type of variable does not generate a memory access item. GCC also uses pseudo-registers to store temporary computation results. Hence, an access to a temporary variable does not cause a memory access item to be generated. Since GCC does not assign pseudo-registers to global variables and aggregate variables, they generate memory access items.

There are some memory access items produced in GCC that do not correspond to any actual variable accesses in the source program. These memory accesses are used for parameter and return value passing in subroutine calls. The actual number of parameter registers available is machine dependent. For each subroutine, GCC uses the parameter registers to pass as many parameters as possible, and then uses the stack to pass the remaining parameters. Hence, at a subroutine call site, if a memory value is passed to the subroutine via a parameter passing register, a memory read is used to load the value from memory to the register. If a register value is passed to the subroutine via the stack, however, a memory write is generated to store the value to the stack. Similarly, at a subroutine entry point, if a memory value is passed into the subroutine via a register, a memory write is generated to store the value to the memory. If a register value is passed into the subroutine via the stack, though, a memory read is again used to load the value from the memory to the register.

A subroutine return value can also generate memory accesses that do not correspond to any variable accesses in the source program. One register is available to handle return values. When the returned value is a structure, the address of the structure is stored in that register at the subroutine call site. In this case, the return statement generates a memory write to store the return value to the memory location indicated by the value return register. If the return value is a scalar, the value return register directly carries the value, so no memory access is generated.

3.1.2 HLI table construction

The HLI table construction phase traverses the SUIF IR twice. The first traversal creates a line table for each routine by collecting the memory access item information from the SUIF annotations. The first traversal also creates a hierarchical region structure for each routine, and groups all the memory access items in a region into equivalent access classes according to the following rules:

1. All memory accesses to the same variable are grouped into one equivalent access class.
2. A pointer reference which definitely refers to a variable is put into the equivalent access class of that variable.
3. An equivalent access class is created for each pointer reference that may refer to multiple locations.
4. An array reference also occupies an equivalent access class. Subsequent data dependence analysis will merge two dependent equivalent access classes with a loop-carried data dependence distance of zero.

The second traversal of the IR visits the hierarchical region structure of each routine in a depth-first fashion. At each node, it gathers the loop-carried data dependence information for each pair of equivalent access classes and calculates the alias relationship between each pair of equivalent access classes. All of the information propagates from the bottom up. If the SUIF data dependence test for a pair of array equivalent access classes in a region returns zero distance, the two equivalent access classes are merged. Otherwise, the test results are stored into the loop-carried data dependence table. Then, all the pointer references that may refer to multiple locations are determined. An alias relationship is created between the equivalent access class for each pointer reference and the equivalent access class to which the pointer reference may refer. Next, the equivalent access class information and alias information is propagated to the immediately enclosing parent region according to the following rules:

1. If an equivalent access class is for a scalar reference, check to see if the parent region has an equivalent access class for that scalar. If yes, add the ID of the equivalent access class in the sub-region to the equivalent access class in the parent region. Otherwise, create a new equivalent access class in the parent region.
2. Create an equivalent access class in the parent region to enclose all the equivalent access classes in the sub-region that refer to accesses to the same array.
3. For each pointer reference equivalent access class in the sub-region, create an equivalent access class in the parent region.
4. If two equivalent access classes in the sub-region are aliases, and the two equivalent access classes are enclosed by two different equivalent access classes in the parent region, then the two equivalent access classes in the parent region are also aliases.

At the completion of these two phases, the HLI is ready to be exported to the back-end.

3.2 Back-end implementation

3.2.1 Importing and mapping HLI into GCC

The HLI file is read on demand as GCC compiles a program function by function. This approach obviates the need to keep all of the HLI in memory at the same time, relieving the memory space requirements on the back-end. The imported information is stored in a separate, generic data structure to enhance portability. Mapping the items listed in the line table onto memory references in the GCC RTL chain is straightforward since the HLI generation in the front-end (Section 3.1.1) follows the GCC rules for memory reference generation when creating the line table. A hash table is constructed as the mapping procedure proceeds to allow GCC quick access to the HLI. A memory reference in GCC, or other back-end compilers, can be represented as a 2-tuple: (IRInsn, RefSpec), where IRInsn specifies an RTL instruction and RefSpec identifies a specific memory access among possibly several memory accesses in the instruction. The hash table forms a mapping between each item and the corresponding (IRInsn, RefSpec) pair.

3.2.2 Using HLI

Information in the HLI can be utilized by a back-end compiler in many ways. Accurate data dependence information will allow aggressive scheduling of a memory reference across other memory references, for example. Additionally, loop-carried data dependence information is indispensable to implement a cyclic scheduling algorithm such as software pipelining [16]. In loop invariant code removal, a memory reference can only be moved out of a loop when it is guaranteed that no other remaining memory reference in the loop can possibly alias

```
/* remove from the hash table all the expressions with a mem. ref.
   clobbered by a function call (call, call_spec) */
static void invalidate_memory_clobbered (call, call_spec)
{
 for (i = 0; i < NBUCKETS; i++)
   for (p = table[i]; p; p = next) {
     next = p->next_same_hash;
     for each mem. ref. (mem, mem_spec) in p
       switch (HLI_GetCallAcc (mem, mem_spec, call, call_spec) {
         case HLI_CALL_MOD:
         case HLI_CALL_REFMOD:
           remove_from_table (mem, mem_spec);
   ... }
... }
}
```

Figure 4: Using call REF/MOD information to aid GCC's Common Subexpression Elimination optimization.

the memory reference. High-level program structure information, such as the line type and the parent line, may provide hints to guide heuristics for efficient code scheduling.

To provide a common interface across different back-ends, HLI stored in the data structures can be retrieved only by using a set of query functions. There are five basic query functions (listed in Appendix A.1) which can be used to construct more complex query functions [5]. There are another set of *utility functions* that simplify the implementation of the user-accessible query functions and maintenance functions (see Section 3.2.3) by hiding the low-level details of the target compiler. Rather than describing the internals of the query functions, two examples are given in this section to show how the query functions can be used in GCC.

In GCC's *Common Subexpression Elimination* (CSE) pass, subexpressions are stored in a table as the program is compiled, and, when they appear again in the code, the already calculated value in the table can directly replace the subexpression. Without interprocedural information, however, all the subexpressions containing a memory reference will be purged from the table when a function call appears in the code since GCC pessimistically assumes that the function call may change any memory location. In Figure 4, an HLI query function to obtain call REF/MOD information is used to remedy the situation by selectively purging the subexpressions on a function call.

The example in Figure 5 shows how the HLI provides memory dependence information to the GCC instruction scheduler. This example is used in Section 4.2 to measure the effectiveness of using HLI to improve the code scheduling pass.

3.2.3 Maintaining HLI

As GCC performs various optimizations, some memory references can be deleted, moved, or generated. These changes break the links between HLI items and GCC memory references set up at the mapping stage, requiring appropriate actions to reestablish the mapping to respond to the change. Further, some of the HLI tables may need updating to maintain the integrity of the information. Typical examples of such optimizations include:

- The CSE pass, where an item may be deleted. The corresponding HLI must then be deleted.
- In the loop invariant removal optimization, an item may be moved to an outer region. The HLI item must be deleted and inserted in the outer region. All the HLI tables must be updated accordingly.

```
/* given a mem. write A and a mem. read B, add a dependence
   edge if there is a true dependence from A to B */
{
  int gcc_value, hli_value, final_value;
  HLI_EquivAccType hli_qresult;

  gcc_value = true_dependence (A, B);  /* GCC query function */
  hli_qresult = HLI_GetEquivAcc (A, B);  /* HLI query function */
  hli_value = (hli_qresult != HLI_NONE);
  final_value = flag_use_hli ? gcc_value * hli_value : gcc_value;
  if (final_value)
    add_dependence (A, B, DEP_TRUE);
}
```

Figure 5: Using equivalent access and alias information for dependence analysis in GCC's instruction scheduling pass.

```
/* construct LCDD info. for the unrolled loop A', based on
   the info. about the original loop A */
for each LCDD [item i, item j, d, t] between item i and j with
                         distance d and type t in A {
  /* K is the unroll factor */
  /* item[a] b is the item b in the a'th unrolled loop */
  for all u (0 <= u < K) {
    if (floor ((u+d)/K) == 0)
      HLI_MergeEquivAcc (item[u] i, item[(u+d)%K] j);
    else
      HLI_AddLCDD (item[u] i, item[(u+d)%K] j, floor((u+d)/K), t);
  }
}
```

Figure 6: Updating the LCDD information for loop unrolling.

- In loop unrolling, the loop body is duplicated and preconditioning code is generated. The entire HLI components (tables) must be reconstructed using old information, and the old information must be discarded.

The HLI maintenance functions (listed in Appendix A.2) have been written to provide a means to update the HLI in response to these changes. The functions allow a back-end compiler to generate or delete items, inherit the attributes of one item to another, insert an item into a region, and manipulate the HLI tables. Changes such as the CSE or loop invariant code removal, call for a relatively simple treatment – either deleting an item, or generating, inheriting, moving, and deleting an item. Loop unrolling, however, requires more complex steps to update the HLI. First, new items need be generated as the target loop body is duplicated multiple times. The generated items are inserted in different regions, based on whether they belong to the new (unrolled) loop body or the preconditioning code. Data dependence relationships between the new items are then computed using the information from the original loop. An example of updating the HLI tables for the loop unrolling pass is given in the Figure 6 in a pseudo-code format, showing how LCDD information based on the old information can be reconstructed.

Benchmark	Suite	Code size (# of lines)	HLI size (KB)	$\frac{\text{HLI size}}{\text{Code size}}$ (bytes/line)
wc	GNU utilities	972	11	12
008.espresso	CINT92	37074	613	17
023.eqntott	CINT92	6269	99	16
129.compress	CINT95	2235	21	10
mean	–	–	–	13
015.doduc	CFP92	25228	1310	53
034.mdljdp2	CFP92	6905	121	18
048.ora	CFP92	1249	29	24
052.alvinn	CFP92	475	7	15
077.mdljsp2	CFP92	4865	109	23
101.tomcatv	CFP95	780	17	22
102.swim	CFP95	1124	76	69
103.su2cor	CFP95	6759	239	36
107.mgrid	CFP95	1725	35	21
141.apsi	CFP95	21921	442	21
mean	–	–	–	27

Table 1: Benchmark program characteristics.

4 Benchmark Results

Test programs selected from the SPEC92 and SPEC95 benchmark suites [26] and the GNU utilities are used in experiments to determine how effective using the HLI is in improving the optimization potential in GCC's code scheduling pass. Note that this code scheduling is limited to basic blocks and, thus, does not demonstrate the true capability of using HLI.

4.1 Program characteristics

Table 1 lists all of the benchmark programs, both integer and floating-point, showing the number of lines of source code, the HLI size in KBytes, and the ratio of the HLI size to the code size. This ratio shows the average number of bytes needed for the HLI for each source code line. We have only a few integer programs due to current implementation limitations of the SUIF front-end tools.[3]

In general, this table shows that a floating-point program requires more space for the HLI than an integer program, implying that the former tends to have more memory references per line. The relatively large HLI size per source code line in *015.doduc* and *102.swim* is mainly due to a large number of items in nested loops, which cause the alias table and the LCDD table to grow considerably.

[3]Our implementation uses the SUIF parser twice (see Figure 3). After the program foo.c is compiled and optimized by SUIF, the optimized C file foo.opt.c is generated. This code is then used as the input to the HLI generation and GCC. When foo.opt.c is fed into the SUIF parser again for the HLI generation, it causes unrecoverable errors in some cases. We are currently developing a front-end compiler (see Section 6) which will not have such difficulties.

Benchmark	Total # of tests	# of tests per line	GCC result	HLI result	"Combined" result	Reduction rate
wc	113	0.12	40 (35%)	20 (18%)	20 (18%)	50%
008.espresso	4166	0.11	2615 (63%)	1316 (32%)	1006 (24%)	62%
023.eqntott	399	0.06	249 (62%)	191 (48%)	120 (30%)	52%
129.compress	274	0.12	56 (20%)	39 (14%)	37 (14%)	34%
mean	–	0.10	– (41%)	– (25%)	– (21%)	48%
015.doduc	10992	0.44	7712 (70%)	3293 (30%)	2855 (26%)	63%
034.mdljdp2	3013	0.44	1753 (58%)	393 (13%)	265 (9%)	85%
048.ora	363	0.29	52 (14%)	79 (22%)	34 (9%)	35%
052.alvinn	48	0.10	47 (98%)	20 (42%)	20 (42%)	57%
077.mdljsp2	2854	0.59	1765 (62%)	413 (14%)	271 (9%)	85%
101.tomcatv	286	0.37	191 (67%)	29 (10%)	14 (5%)	93%
102.swim	872	0.78	833 (96%)	83 (10%)	80 (9%)	90%
103.su2cor	4192	0.62	3549 (85%)	1602 (38%)	1453 (35%)	59%
107.mgrid	517	0.30	368 (71%)	330 (64%)	311 (60%)	15%
141.apsi	22347	1.02	8031 (36%)	6375 (29%)	5399 (24%)	33%
mean	–	0.42	– (59%)	– (26%)	– (19%)	54%

Table 2: Using the HLI in GCC's dependence checking routines can substantially reduce the number of dependence arcs that must be inserted into the DDG.

4.2 Augmenting GCC dependence analysis

Instruction scheduling is an important code optimization in an optimizing back-end compiler. With this optimization, instructions in a code segment are reordered according to a set of rules to minimize the number of cycles required to execute the code. A crucial step in instruction scheduling is to determine whether or not there is a dependence between two memory references when at least one is a store (memory write). Accurately identifying such dependences can reduce the number of edges in the data dependence graph, thereby giving the scheduler more freedom to move instructions around to improve the quality of the scheduled code.

GCC supplies a set of functions to compute memory dependences for the instruction scheduler. The functions utilize information about memory references collected in earlier passes to perform the dependence analysis. Given two memory references A and B, GCC determines that there is no dependence between A and B if: (1) either A or B is constant (unchanging within a function), or (2) A is in a structure, but B is not, or (3) A is addressed by the frame pointer, and B is a static variable, or (4) A and B are static variables with different addresses.

HLI can potentially enhance the GCC instruction scheduling optimization by providing more accurate memory dependence information when GCC would otherwise have to make a conservative assumption due to its simple dependence analysis algorithm. For the programs tested, Table 2 shows the total number of dependence queries (i.e. do A and B refer to the same memory location?) made in the first instruction scheduling pass of GCC, the average number of queries for each source code line, the number of times the GCC analyzer answers yes, meaning that it must assure there is dependence, the number of times HLI answers yes, and lastly, the number of times both GCC and HLI answer yes. Since a number in the table denotes the number of dependence edges inserted into the DDG, the smaller the number, the more accurate the corresponding analyzer. The last column shows the reduction rate in the number of dependence edges for each program due to the use of the HLI.

The result shows that using HLI could reduce the number of dependence edges significantly, by an average of 48% for the integer programs and 54% for the floating-point programs. Four floating-point programs – *034.mdljdp2*, *077.mdljsp2*, *101.tomcatv*, and *102.swim* – exhibited a reduction of over 80% in the number of dependence edges. The reduction in dependence edges allows more aggressive instruction scheduling, potentially decreasing the execution times of these programs. These results confirm that the data dependence information extracted by the front-end analysis is very effective in disambiguating memory references in the back-end compiler.

Note that the numbers in the *HLI result* and *"Combined" result* columns in Table 2 are not the same in most of the cases, which means that there is room for additional improvement in the HLI. Current shortcomings in generating the HLI include – (1) the currently implemented front-end algorithms, such as the array data dependence analysis and the pointer analysis, are not as aggressive as possible, (2) there are miscellaneous GCC code generation rules that the current implementation of the HLI generation has not considered, which produces *unknown* dependence types between some memory references, and (3) the maintenance of the HLI in GCC has not been completely developed. In any case, it is guaranteed that the correctness of HLI is not affected since conservative assumptions are made when the mapping is done, when queries are made, and when the HLI is maintained. The values of the *HLI result* and the *"Combined" result*, as well as the difference between them, are, therefore, all expected to become smaller as more aggressive front-end algorithms are developed and the current implementation limitations are overcome.

4.3 Impact on program execution times

To study the performance improvement attributable to using HLI in GCC's instruction scheduling optimization pass, execution times of the benchmark programs, compiled both with and without HLI, were measured on two real machines. Machine configuration A contains a pipelined processor (MIPS R4600) with a 100 MHz clock, 16 KB of on-chip data cache, 16 KB of on-chip instruction cache, and 64 MB of main memory. Machine configuration B has a superscalar processor (MIPS R10000) with a 196 MHz clock, a 32 KB on-chip data cache, a 32 KB on-chip instruction cache, a 2 MB unified off-chip second-level cache, and 512 MB of interleaved main memory.

All the programs were compiled with GCC version 2.7.2.2 with the -O2 optimization flag. Each program execution used the reference input. The input to the program *wc* is 62 MBytes of C source codes. Programs were run three times to obtain an average execution time. The variance in execution times was negligible. Table 3 summarizes the results of the experiment.

Three programs achieved a noticeable speedup of 5% or more on machine A, with five programs (including the previous three) achieving similar results on machine B. Two programs, *034.mdljdp2* and *077.mdljsp2*, obtained remarkable speedups of over 40% on machine B. Note that a large reduction in dependence edges, as shown in Table 2, does not always result in a large execution time speedup, as can be seen in *101.tomcatv*, for instance. This is due to a limitation of the GCC instruction scheduler, which schedules instructions only within basic blocks. This may rather severely limit the range within which an instruction can move. Further, to obtain a substantial speedup, the most frequently executed basic blocks must be optimized well.

The integer programs achieved relatively small speedups compared to the floating-point programs. It is known that the basic blocks in integer programs are usually very small, containing only 5 – 6 instructions on average, and it is likely that each basic block contains few memory references. This is indirectly evidenced by comparing the number of dependence queries made per line (see Table 2). Typically, an integer program requires

	Machine configuration A (MIPS R4600)			Machine configuration B (MIPS R10000)		
Benchmark	Without HLI	With HLI	Speedup	Without HLI	With HLI	Speedup
wc	12.4	12.4	1.00	4.0	4.0	1.00
008.espresso	47.4	47.3	1.00	11.5	11.5	1.00
023.eqntott	19.2	19.1	1.01	6.1	5.8	1.05
129.compress	1642.0	1546.8	1.06	342.9	321.8	1.07
mean	–	–	1.02	–	–	1.03
015.doduc	70.8	71.1	1.00	13.1	12.7	1.03
034.mdljdp2	362.0	334.7	1.08	68.7	48.4	1.42
048.ora	304.8	304.8	1.00	39.1	39.1	1.00
052.alvinn	64.1	63.6	1.01	17.0	16.7	1.02
077.mdljsp2	235.0	198.2	1.19	41.7	26.2	1.59
101.tomcatv	66.8	66.7	1.00	16.4	16.3	1.01
102.swim	51.4	49.7	1.03	12.5	12.0	1.04
103.su2cor	32.5	31.9	1.02	8.1	7.5	1.08
107.mgrid	183.0	183.0	1.00	68.3	67.6	1.01
141.apsi	447.6	449.1	1.00	59.6	58.9	1.01
mean	–	–	1.03	–	–	1.11

Table 3: The execution times of programs compiled using HLI in GCC's code scheduling pass are smaller than those compiled using GCC's standard dependence analysis tests, resulting in a speedup attributable to using the HLI.

fewer than half the number of dependence tests needed by a floating-point program. Since the GCC instruction scheduler schedules instructions only within basic blocks, the integer programs do not benefit much from using the HLI.

Comparing the different processor types, machine B produces higher speedups than machine A, with each speedup on machine B being equal to or higher than the corresponding speedup on machine A. There are two primary reasons for this difference. First, machine B, which is a superscalar processor, can issue up to four instructions in a cycle. Since a stall caused by a memory reference eliminates the opportunity to execute several instructions, the memory performance has a greater impact on the execution time than on machine A. Second, machine B incorporates a load/store queue where all memory references are queued in the program order before their execution. A load instruction in the queue will not be issued to the memory system until all preceding store instructions in the queue are known to be independent of the load. Therefore, if the memory instructions are scheduled well at compile time, load instructions can issue more freely at run time in this machine, thereby decreasing the execution time.

5 Related Work

Traditionally, parallelizing compilers and optimizing compilers for uniprocessors have been largely two separate efforts. Parallelizing compilers perform extensive array data dependence analysis and array data flow analysis in order to identify parallel operations. Based on the results, a sequential program is transformed into a parallel program containing program constructs such as DOALL loops, which are loops whose iterations can be executed in parallel. Alternatively, a parallelizing compiler may insert a compiler directive before a sequential loop to indicate that the loop can be executed in parallel. Several research parallelizing Fortran compilers, including Parafrase [15], PFC [1], Parafrase-2 [21], Polaris [3], Panorama [11], and PTRAN [22], and commercial Fortran compilers, such as KAP [13, 14] and VAST [28], have taken such a source-to-source approach.

Computer vendors generally provide their own compilers to take a source program, which has been parallelized by programmers or by a parallelizing compiler, and generate multithreaded machine code, i.e., machine code embedded with thread library calls. These compilers usually spend their primary effort on enhancing the efficiency of the machine code for individual processors. Multithreaded programs may also be written in a relatively high-level language such as C. The SUIF tool [29], for example, provides an option to transform a Fortran program or a C program into a multithreaded C program. Once the thread assignment to individual processors has been determined, parallelizing compilers have little control over the execution of the code by each processor. They do not schedule parallel operations on multiple functional units and they do not allocate registers because their output is just another source program.

Over the past years, both *machine independent* and *machine specific* compiler techniques have been developed to enhance the performance of uniprocessors. The former includes well-known techniques such as common subexpression elimination, dead-code removal and constant folding [19, 6]. Recent important machine-specific techniques primarily aim at improving instruction-level parallelism and register usage. *Trace scheduling* and *software pipelining* are among the most prominent instruction scheduling techniques for efficiently utilizing multiple functional units within a single processor. These techniques have been implemented in research prototypes such as ELI [10, 20] and IMPACT [12], as well as production compilers, such as the Cydra-5 [7] and Multiflow compilers [17]. Current compiler techniques for uniprocessor performance enhancement, both machine independent and machine specific, primarily rely on dataflow analysis for symbolic registers or simple scalars that are not aliased. Advanced data dependence analysis and data flow analysis regarding array references and pointer dereferences are generally not available to current uniprocessor compilers. The publically available GCC [25] and LCC [9] compilers exemplify the situation – both compilers maintain low-level intermediate representations of the input programs, keeping no high-level program constructs for array data dependence analysis and pointer-structure analysis.

With the increased demand for instruction-level parallelism, the importance of incorporating high-level analysis into uniprocessor compilers has been generally recognized. Recent work on pointer and structure analysis aims at accurate recognition of aliases due to pointer dereferences and due to pointer arguments passed to called routines [8, 30]. Experimental results in this area have been limited to reporting the accuracy of recognizing aliases. Compared with these studies, this paper presents new data showing how high-level array and pointer analysis can improve dataflow analysis in GCC, a common uniprocessor compiler.

One should note that source-to-source program transformations based on high-level program analysis may indirectly improve uniprocessor performance. For example, array data dependence analysis can enable *scalarization* of array references, which provides uniprocessor compilers with more opportunities for enhancing register allocation [4]. Numerous studies exist in the literature that discuss source-level program transformations to improve the memory performance on uniprocessors (see [18], for example). There have been continued efforts to incorporate uniprocessor parameters and knowledge about low-level code generation strategies into the high-level decisions about program transformations. The ASTI optimizer for the IBM XL Fortran compilers [23] is a good example. Nonetheless, the register allocator and instruction scheduler of the uniprocessor compiler still lacks direct information about data dependences concerning complex memory references.

New efforts on integrating parallelizing compilers with uniprocessor compilers also have emerged recently. The SUIF tool [29], for instance, maintains a high-level intermediate representation that is close to the source program to support high-level analysis and transformations. It also maintains a low-level intermediate representation that is close to the machine code. As another example, the Polaris parallelizing compiler has recently incorporated a low-level representation to enable low-level compiler techniques [2]. Nonetheless, results showing how high-level analysis benefits the low-level analysis and optimizations are largely unavailable today. Our effort has taken a different approach by providing a mechanism to transport high-level analysis results to uniprocessor compilers, using a format that is relatively independent of the particular parallelizing compiler and the particular uniprocessor compiler. We believe that such a mechanism makes it relatively easy to integrate any existing parallelizing compiler with any existing uniprocessor compiler.

6 Conclusions and Future Work

The development of a new compiler, especially one intended to automatically parallelize sequential application programs, is a complex and difficult task. As a result, most parallelizing compilers have been split into a front-end that performs high-level program analysis and code transformation, and a back-end that focuses on low-level, machine-specific optimizations. This front-end/back-end split simplifies the development of the entire compiler, and reduces the total amount of data that must be preserved throughout the compilation process. Because of this split, however, the back-end no longer has access to the high-level dataflow and program structure information. The loss of this information then limits the scope and type of optimizations that the back-end can perform.

To address this problem, a few compiler projects (see Section 5) have begun to use information about low-level code generation and machine details to improve the application of high-level program transformations. They also are beginning to use high-level information to guide instruction-level optimizations. Instead of integrating the front-end and back-end into a single compiler, however, we have taken a different approach that provides a mechanism to export the results of high-level program analysis from the front-end to a standard back-end compiler. This high-level information is transferred using a well-defined format (HLI) that condenses the high-level information to reduce the total amount of data that must be transferred. Additionally, this format is relatively independent of the particular front-end parallelizing compiler and the specific back-end compiler.

We have demonstrated the effectiveness of this approach by implementing it into the SUIF front-end and the GCC back-end compilers. Our experiments with the SPEC benchmarks show that using this information in the code scheduling pass of GCC substantially reduces the number of dependence arcs that must be inserted into the data dependence graph. The increased flexibility provided by this reduction allowed the code scheduler to improve execution time performance by up to 59% compared to using only the low-level information normally available to the back-end.

We believe that the HLI mechanism proposed in this paper makes it relatively easy to integrate any existing front-end parallelizing compiler with any existing back-end compiler. In fact, we are currently developing a new front-end parallelizing compiler[4] that will use the HLI mechanism to export high-level program information to the same GCC back-end implementation used in these experiments.

[4] See http://www.cs.umn.edu/Research/Agassiz/.

Acknowledgment

This work was supported in part by the National Science Foundation under grant nos. MIP-9610379 and CDA-9502979; by the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, and a gift from the Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. Stephen Schwinn is currently with the IBM Corp., Rochester, MN.

References

[1] J. R. Allen and K. Kennedy. "Automatic Translation of FORTRAN Programs to Vector Form," in *ACM Transactions on Programming Languages and Systems*, 9(4): 491 – 542, October 1987.

[2] E. Ayguade, C. Barrado, J. Labarta, D. Lopez, S. Moreno, D. Padua, and M. Valero. "A Uniform Internal Representation for High-Level and Instruction-Level Transformations," *Technical Report 1434*, CSRD, Univ. of Illinois at Urbana-Champaign, 1994.

[3] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. "Parallel Programming with Polaris," in *IEEE Computer*, December 1996.

[4] D. Callahan, S. Carr, and K. Kennedy. "Improving Register Allocation for Subscripted Variables," in *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 53 – 65, June 1990.

[5] S. Cho and Y. Song. "The HLI Implementor's Guide (v0.1)," *Agassiz Project Internal Document*, http://www.cs.umn.edu/Research/Agassiz/memo.html, September 1997.

[6] F. C. Chow. A Portable Machine-Independent Global Optimizer – Design and Measurements, *Ph.D. Thesis*, Stanford University, December 1983.

[7] J. C. Dehnert and R. A. Towle. "Compiling for the Cydra 5," in *The Journal of Supercomputing*, 7(1/2): 181 – 227, 1993.

[8] M. Emami, R. Ghiya and L. J. Hendren. "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242 – 256, June 1994.

[9] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

[10] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures, MIT Press, Cambridge, Mass., 1986.

[11] J. Gu, Z. Li, and G. Lee. "Experience with Efficient Array Data Flow Analysis for Array Privatization," in *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, June 1997.

[12] W. W. Hwu *et al.* "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," in *The Journal of Supercomputing*, 7(1/2): 229 – 248, 1993.

[13] KAP User's Guide, *Technical Report (Document No. 8811002)*, Kuck & Associates, Inc.

[14] KAP for IBM FORTRAN, User's Guide Version 3.3, *Technical Report (Document No. 8811002)*, Kuck & Associates, Inc.

[15] D. J. Kuck *et al.* "The Structure of an Advanced Vectorizer for Pipelined Processors," in *Proceedings of the 4th International Computer Software and Application Conference*, pp. 709 – 715, October 1980.

[16] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[17] P. G. Lowney *et al.* "The Multiflow Trace Scheduling Compiler," in *The Journal of Supercomputing*, 7(1/2): 51 – 142, 1993.

[18] K. S. McKinley, S. Carr, and C.-W. Tseng. "Improving Data Locality with Loop Transformations," in *ACM Transactions on Programming Languages and Systems*, Vol. 18, pp. 423 – 453, July 1996.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.

[20] A. Nicolau. Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers, *Ph.D. Thesis*, Yale University, 1984.

[21] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. A. Schouten. "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," in *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[22] V. Sarkar. The PTRAN Parallel Programming System, *Parallel Functional Programming Languages and Compilers*, B. Szymanski, Ed., ACM Press, pp. 309 – 391, 1991.

[23] V. Sarkar. "Automatic Selection of High-Order Transformations in the IBM XL FORTRAN Compilers," in *IBM Journal of Research and Development*, 41(3): 233 – 264, May 1997.

[24] S. J. Schwinn. "The HLI Interface Specification for Back-End Compilers (v0.1)," *Agassiz Project Internal Document*, http://www.cs.umn.edu/Research/Agassiz/memo.html, September 1997.

[25] R. M. Stallman. Using and Porting GNU CC (for GCC version 2.7), Free Software Foundation, Cambridge, MA, June 1995.

[26] The Standard Performance Evaluation Corporation, http://www.specbench.org.

[27] J.-Y. Tsai. "High-Level Information Format for Integrating Front-End and Back-End Compilers (v0.2)," *Agassiz Project Internal Document*, http://www.cs.umn.edu/Research/Agassiz/memo.html, March 1997.

[28] VAST-2 for XL FORTRAN, User's Guide, Edition 1.2, *Technical Report (Document No. VA061)*, Pacific-Sierra Research Corporation, 1994.

[29] R. P. Wilson *et al.* "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," in *ACM SIGPLAN Notices*, 29 (12): 31 – 37, December 1994.

[30] R. P. Wilson and M. S. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs," in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1 – 12, June 1995.

A HLI Query and Maintenance Functions

A.1 HLI query functions

The query functions are used to retrieve the stored HLI information. The following functions have been implemented:

- HLI_GetEquivAcc determines whether two arbitrary memory references access the same memory location.
- HLI_GetCallAcc determines whether one memory location is referenced and/or modified within a function call.
- HLI_GetLCDD determines whether two memory references have a loop-carried data dependence and provides the type of the dependence if any.
- HLI_GetLine determines the line type for a particular line.
- HLI_GetParentLine determines the parent line of a particular line.

A.2 HLI maintenance functions

The maintenance functions are used to update the stored HLI information when the back-end compiler performs an optimization which will affect the accuracy or correctness of the HLI information. The following functions have been implemented:

- HLI_New creates a new, empty HLI entry for a memory access or call insn.
- HLI_Inherit makes one HLI entry to be identical to another, i.e., to be in the same equivalent access class.
- HLI_Move moves an item from one location to another (possibly across region boundaries).
- HLI_Delete marks an item as 'deleted'.
- HLI_Undelete clears the 'deleted' mark on an item.
- HLI_MergeEquivAcc merges two equivalent access classes.
- HLI_SplitEquivAcc splits one equivalent access classes into two.
- HLI_AddLCDD adds a loop-carried data dependence entry.
- HLI_RemoveLCDD removes a loop-carried data dependence entry.
- HLI_AddAlias adds an alias entry.
- HLI_RemoveAlias removes an alias entry.
- HLI_AddCallAcc adds an entry to the function call REF/MOD table.
- HLI_RemoveCallAcc removes an entry from the function call REF/MOD table.