Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 99-016

Design & Implementation of a Framework for Integrating Front-End & Back-End Compilers

Pen-chung Yew, Zhiyuan Li, Yonghong Song, Jenn-yuan Tsai, and Bixia Zheng

April 20, 1999

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **20 APR 1999**	2. REPORT TYPE	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Design & Implementation of a Framework for Integrating Front-End & Back-End Compilers**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Army Intelligence Center & Fort Huachuca,Fort Huachuca,AZ,85613**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**27**	

Design and Implementation of a Framework for Integrating Front-End and Back-End Compilers

Authors:
Sangyeun Cho[*], Qing Zhao[*], Bixia Zheng[*], Jenn-Yuan Tsai[†], Yonghong Song[‡], Zhiyuan Li[‡], David J. Lilja[§], Pen-Chung Yew[*]

Corresponding Author:
Qing Zhao
Dept. of Computer Science and Engineering, Univ. of Minnesota, 200 Union St. S.E.
Minneapolis, MN 55455, USA

Tel: (612) 624-8545 Fax: (612) 625-0572 E-mail: zhao@cs.umn.edu

Abstract:
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 substantially for the benchmark programs studied, which provides greater flexibility to GCC's code scheduling pass, common subexpression elimination pass, loop invariant code removal pass and register local allocation pass. Even with GCC's optimizations 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 a MIPS R10000 processor.

Keywords: data dependence, memory disambiguation, instruction-level parallelism, program optimization, parallelizing compiler.

[*]Department of Computer Science and Engineering, University of Minnesota, Minneapolis.
[†]Hewlett-Packard Company, Cupertino, CA.
[‡]Department of Computer Science, Purdue University, West Lafayette, IN.
[§]Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis.

1 Introduction

High-performance microprocessors increasingly rely on parallel operations to speed up program execution. Recent superscalar processors fetch multiple instructions, dynamically find independent instructions from a set of *reservation stations* (or a *window* of instructions), and issue them in parallel to multiple function units [12, 13, 14]. Extensive research is under way to increase the exploitable *instruction level parallelism* (ILP) in a program and to widen the issue bandwidth from 4 to 8 instructions per cycle, or even up to 16 [19, 23]. Additionally, researchers have begun to explore *thread-level parallelism* in which multiple threads of instructions can be simultaneously fetched by different thread-execution units for processing [22, 27, 30, 31]. These thread-execution units are more tightly coupled than processors in a multiprocessor system in that the order of instruction dispatching and retiring is tightly synchronized among different units.

The trend towards higher-level hardware parallelism imposes a severe demand on compilers to analyze and transform programs more aggressively to uncover parallelism since it becomes increasingly more difficult for hardware to extract parallelism alone. Traditionally, compilers for microprocessors use a low-level *intermediate representation* (IR) of the program to analyze dependences and the data flow between operations. Such low-level IR, or LIR, normally lacks information regarding naming, types and aliases concerning arrays, and other high-level data structures. Since references to such variables become a series of indirect references using simple scalar variables or so-called *symbolic registers*, array subscripts and pointer expressions all become obscured.

This type of LIR has served microprocessor compilers reasonably well so far mainly because sufficient ILP can often be exploited among scalar operations within a relatively narrow program scope. Such parallelism can either be detected by hardware without code transformation or it can be exposed relatively simply through code motion done by the *instruction scheduler* in the compiler. To uncover additional parallel operations to feed the increasing hardware parallelism in the future, however, the compiler needs to analyze a wider program scope and higher-level data structures. The compiler must perform high-level program semantic analysis regarding arrays and pointers, such as analysis of data dependences, aliases, data flow, loop level parallelism, and summary use-modification information for procedure calls. Only a high-level IR (HIR) contains the necessary abstract syntax information to support this extensive analysis.

The LIR must continue to exist since the low-level machine operations are the instruction scheduling target. Indeed, some low-level operations may not even have a direct equivalent in the HIR. Hence, the problem becomes one of passing high-level semantic information from the HIR to the LIR.

We have designed and implemented a format, called *High-Level Information (HLI)* [4], to facilitate the propagation of high-level semantic information from the HIR to the LIR. Several considerations have influenced our design:

- **Transportability**: The HLI can be exported from a high-level analyzer, such as those used in sophisticated parallelizing compilers, to a microprocessor compiler that does not contain HIR and thus lacks a high-level analyzer.

- **Hierarchy**: The information regarding data dependences and aliases are organized in a hierarchy corresponding to the loop structures in the program. This reduces the complexity of the represented information and makes the access to such information easier for a back-end compiler.

- **Flexibility**: The HLI information can be updated if the program is modified after the HLI is produced, as occurs with many backend optimizations, such as statement reordering. We have implemented and experimented with a maintenance utility for the HLI to perform this updating.

In the remainder of the paper, Section 2 presents the formal definition of the HLI format, showing what information is extracted using the HIR and how it is condensed and passed to the LIR. Section 3 then describes our prototype implementation of this HLI within the SUIF parallelizing compiler and the GCC compiler back-end. Experiments with the SPEC benchmark programs [29] are presented in Section 4, showing how the use of the high-level information provided by SUIF improves the dependence information available to GCC and the execution time of benchmark programs. Some 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. As shown in Figure 1, an HLI

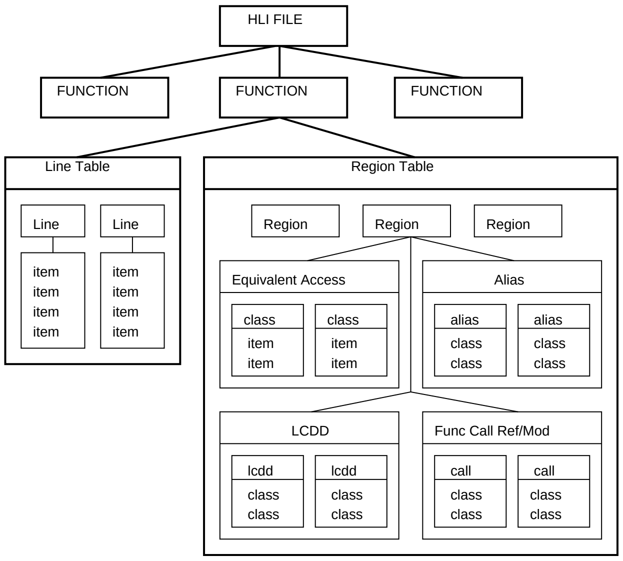


Figure 1: Top-level layout of an HLI file.

file contains a number of HLI entries. Each HLI entry corresponds to a program unit in the source file and contains two major tables, a *line table* and a *region table*, as described below.

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.

The HLI focuses on memory references and function calls, which are called *items* in the HLI representation. In the line table, each line entry corresponds to a source line of the program unit in the source file, and includes an item list for the line. 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 of the item, which may be a load, a store, or a function call.

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 (LCDD) 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. The HLI deals with such complexity by separating data dependences at different region-levels. In each region, two kinds of memory references are distinguished from each other, namely the *immediate* references, which are not embedded in any subregions, and the *embedded* references, which appear inside subregions. Two immediate references are called *equivalent*, and are placed in an *equivalent access class*, if they refer to the same memory location in the same execution instance of the region body (*i.e.* the same iteration if the region is a loop). Likewise, two embedded references are *equivalent* within the current region if they access the same memory locations. If the memory locations accessed by two references within the current region may overlap but are not necessarily identical, then the two references are said to be *maybe* equivalent and their equivalent access class is marked as *maybe*. Moreover, if the memory locations accessed by two references may overlap in one execution instance of the region body but not in another instance, then the two references are also said to be *maybe* equivalent. All other equivalent references are said to be *definitely* equivalent. Since it is easier to identify definitely-equivalent immediate references than embedded references, we do not place an immediate memory reference and an embedded memory reference in the same equivalent access class.

Equivalent access classes serve two purposes. First, they capture data dependences that are loop independent [1]. If two references, one being a load, belong to the same equivalent access class, then they have a loop-independent data dependence within the corresponding region. If an immediate reference and an embedded reference are not placed in the same equivalent access class, their loop-independent data reference, if any, will be represented by the *alias table* described below. The second purpose of equivalent access classes is to reduce the number of loop-carried data dependences [1] that need to be represented. Within the same region, if a number of sources of loop-carried dependences belong to the same equivalent access class, they can be represented by that single equivalent access class. Likewise, a number of dependence sinks can also be represented by a single equivalent access class.

Since the nesting of all regions in a program unit can be viewed as a tree, the HLI uses a tree program structure to accurately and efficiently describe the relationship among equivalent access classes in different regions. Each leaf represents an individual memory reference and each internal node represents an equivalent access class. Each internal node, associated with a particular region, represents the union of the equivalent classes represented by its children that are associated with the immediate sub-regions. Each equivalent access class has a unique ID and each region has an equivalent access table that identifies all equivalent access classes within that region.

Figure 2 demonstrates the region structure of a procedure and its 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 viewpoint of Region 1, every memory access item inside the procedure is represented by exactly one of those equivalent access classes. For example,

```
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}
    }
}
```

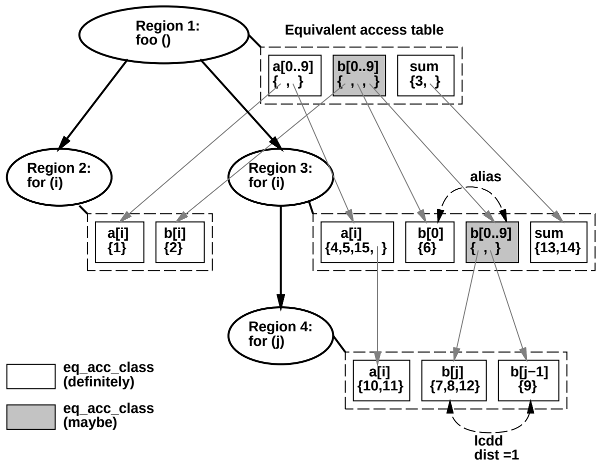


Figure 2: The structure of the regions and equivalent access classes for an example program.

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 the 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

Two memory references referring to two distinct names in the same program unit may actually access the same memory location in the same execution instance of the unit. Such names are known as *aliases*. For convenience, we say that those two memory references are *aliased*. If the two references belong to different equivalent access classes in the same region, we again say that the two classes are *aliased*. An *alias table* is created for each region to describe the possible alias relationships among the equivalent access classes of that region. If two equivalent access classes are marked as aliased, all of the memory access items represented by the two equivalent access classes are considered aliased.

Recall that an immediate reference and an embedded reference are not placed in the same equivalent access class in the same region. If they may access the same memory locations in that region, they are also marked as aliased. Note that *aliasing* is a binary relation between two equivalent classes. A and B being aliases and B and C being aliases does not imply that A and C must be aliases. This is the primary reason that the HLI does not place all aliased references in a single equivalent access class. For a loop region, the alias table only describes the alias relationships among the equivalent access classes within a loop iteration. Loop carried data dependences are described in the LCDD table.

In the example 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 (LCDD) table

If the region is identified as a loop, the LCDD table will list all of the LCDDs at that loop level. 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 *definite* or *maybe*. 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 LCDD is between equivalent access classes *b[j]* and *b[j-1]* in Region 4. The distance of this LCDD 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 whose referenced memory locations may

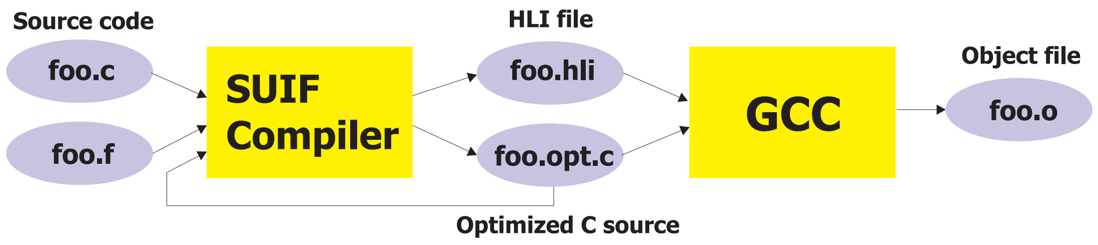


Figure 3: Overview of the HLI implementation using the SUIF front-end and GCC back-end compilers.

be referenced or modified by the called function. For function calls inside 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 whose referenced memory locations may be referenced or modified by the function calls inside the sub-region. With this 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 A Prototype Implementation

A version of the HLI described in the previous section has been implemented in the SUIF parallelizing compiler [33] and the GCC back-end compiler [28]. 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. Figure 3 shows an overview of our HLI implementation [4] 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 and assigns a unique number (ID) to each item. The memory access items for a source line, ordered by the ID, can be one-to-one matched to the memory reference instructions in the GCC RTL chain[1] for the same line. These items are annotated in the SUIF expression nodes to be passed to the TBLCONST phase.

[1]RTL (Register Transfer Language) is an intermediate representation used by GCC that resembles Lisp lists [28]. An RTL chain is the linked list of low-level instructions in the RTL format.

The TBLCONST phase first collects the memory access item information from the SUIF annotation to produce the line table for each program unit. It then generates information for the equivalent access table, alias table, and LCDD table for each region. Because it is dependent on both back-end compiler and the machine, separating the HLI generation into these two phases allows us to reuse the code for TBLCONST across different back-end compilers or target machines.

3.1.1 Memory access item generation (ITEMGEN)

The ITEMGEN phase traverses the SUIF internal representation (IR) to generate 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, the RTL generation rules in GCC must be considered in the HLI generation by SUIF.

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 or a variable used for temporary computation results. An access to this type of variable does not generate a memory access item. 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 into 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. 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 in the MIPS architecture [14] which we target in our implementation. 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 (TBLCONST)

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. It also creates a hierarchical region structure for each routine and groups all the memory access items in a region into equivalent access classes.

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 LCDD 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 LCDD 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 immediate parent region. 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 eliminates 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 ITEMGEN phase in the front-end (Section 3.1.1) follows the GCC rules for memory reference generation. 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.

```
/* 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 *CSE* optimization.

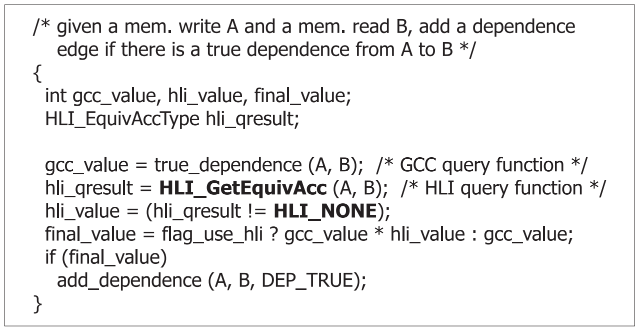
3.2.2 Using HLI

Information in the HLI can be utilized by a back-end compiler in various ways. Accurate data dependence information allows aggressive scheduling of a memory reference across other memory references, for example. Additionally, LCDD information is indispensable for a cyclic scheduling algorithm such as software pipelining [18]. Interprocedural information provides the back-end compiler more freedom to move memory references around function calls. High-level program structure information, such as the line type, may provide hints to guide heuristics for efficient code scheduling.

To provide a common interface across different back-ends, the stored HLI can be retrieved only via a set of query functions. There are five basic query functions that can be used to construct more complex query functions [5]. There are another set of *utility functions* that simplify the implementation of the query and maintenance functions (Section 3.2.3) by hiding the low-level details of the target compiler. 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 can 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 instruction scheduler. It is used in Section 4.2 to measure the effectiveness of using HLI to improve the code scheduling pass.



```
/* 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.

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.
- 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.

```
/* 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.

The HLI maintenance functions have been written to provide a means to update the HLI in response to these changes [5]. 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 update 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 Figure 6.

4 Benchmark Results

To demonstrate how the HLI can be used, we experimented with several programs taken from the SPEC benchmark suite and other sources. High level information is passed from the SUIF front-end to the GCC backend for each program. We then measured the reduction in the number of dependence arcs identified by the GCC memory dependence checking routines when using HLI compared to using only GCC's normal dependence checking capabilites. We also measured the improvement in execution time made possible when GCC used the HLI to improve several of its back-end optimization passes within basic blocks.

Benchmark	Suite	Code size (# of lines)	HLI size (KB)	HLI per line (bytes)
wc	GNU	1262	12	10
023.eqntott	CINT92	7738	108	14
129.compress	CINT95	2252	21	10
008.espresso	CINT92	43505	654	15
Average	–	–	–	12
101.tomcatv	CFP95	719	354	492
102.swim	CFP95	1203	78	64
107.mgrid	CFP95	1794	36	21
103.su2cor	CFP95	6959	248	36
125.turb3d	CFP95	8934	244	27
034.mdljdp2	CFP92	7094	127	18
056.ear	CFP92	4911	91	18
052.alvinn	CFP92	507	7	14
Average	–	–	–	86

Table 1: Benchmark program characteristics.

4.1 Program characteristics

Table 1 lists all of the benchmark programs used in these experiments, 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.[2]

This table shows that, in general, a floating-point program requires two to three times more space for the HLI than an integer program. This suggests that the floating-point programs tend to have more memory references per line. The relatively large HLI size per source code line in *101.tomcatv*, *102.swim* and *103.su2cor* is mainly due to a large number of items in nested loops. This characteristic causes the alias table and the LCDD table to grow substantially compared to the other programs.

[2]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 that will eliminate such difficulties.

4.2 Aiding GCC's dependence analysis

The HLI can potentially enhance several back-end optimizations by providing more accurate memory dependence information when GCC would otherwise have to make a conservative assumption due to its simple dependence analysis algorithm. Four optimizations in GCC were instrumented with the HLI to utilize this more accurate memory dependence information to improve the performance of the resulting code.

Instruction scheduling (*Sched*) is an important code optimization in a back-end compiler. With this optimization, instructions in a code segment are reordered to minimize the overall execution time. A crucial step in instruction scheduling is to determine if there is a dependence between two memory references when at least one is a memory write operation. 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.

In the Common Subexpression Elimination (*CSE*) pass, all subexpressions that reference memory are removed from the table since GCC must assume that any memory reference will change these subexpressions. Distinguishing memory references according to the data dependence information provided by HLI will maintain the subexpressions whose memory is independent with the current memory reference in the table.

In the loop invariant code removal (*Loop*) pass, a memory reference can be moved out of a loop only when there are no other memory references in the loop that could possibly be aliased with the current memory reference. Since HLI will potentially reduce the number of data dependences within the loop, more memory instructions could be taken out of the loop. This then increases the likelihood of a memory operation becoming loop invariant.

In the register local allocation (*Local*) pass, the first step is to find the symbolic registers that are equivalent to a single value throughout the compilation. These are grouped to a single register. In this step, there is one special case–after one register is defined, it is stored to one memory location within a single basic block. If no other memory store operations between the above two instructions could be aliased with the memory location, the two instructions can be combined into one and the register could be eliminated completely. This reduces the total number of instructions and allows the register allocator more degrees of freedom. The more accurate data dependence information provided by the

HLI can help in finding more of these types of instruction pairs.

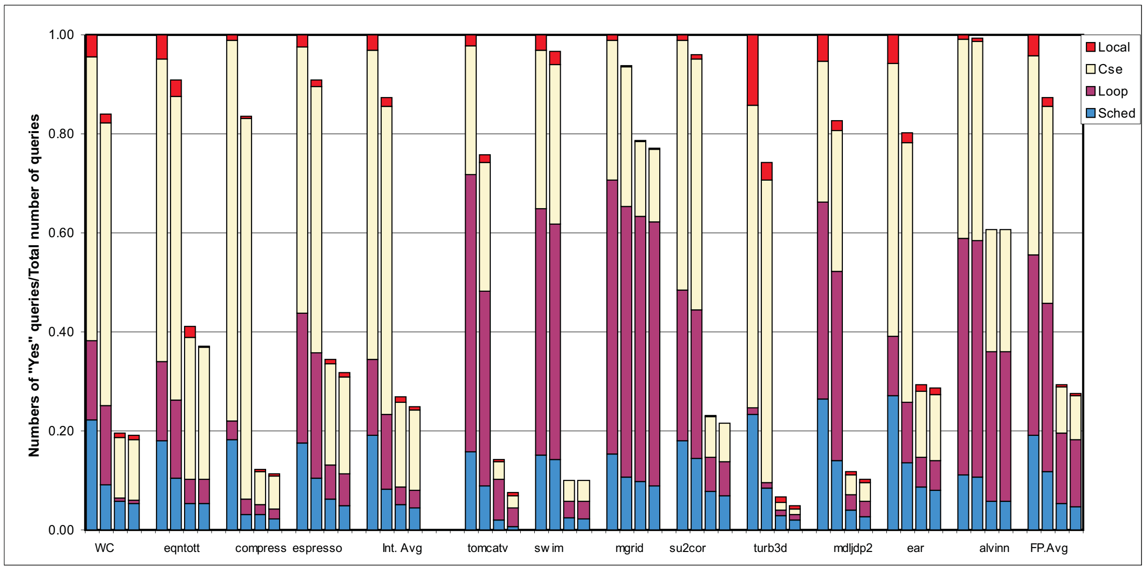


Figure 7: Comparison of GCC results, HLI results and Combined results for data dependence queries. The first bar for each program reports all queries to be true, the next bar is the standard GCC results, the third bar is the HLI results and the final bar shows the combination of standard GCC results with the HLI information. The divisions within each bar show the results of data dependence queries made in each of the four optimization passes.

For the programs tested, Figure 7 compares the total number of dependence queries made (*i.e.*, do A and B refer to the same memory location?), the number of times the GCC analyzer answers yes (meaning that it must assume there is a dependence), the number of times HLI answers yes, and the number of times both GCC and HLI answer yes. The values shown are normalized to the total number of dependence queries. The subsections of each bar correspond to each of the optimization passes studied. Since the height of the bars in the figure corresponds to the number of data dependences that must be assumed, the lower the bar, the more accurate the corresponding analyzer. While the figure shows the normalized number of queries, Table 2 provides the absolute number of total queries and the number of queries made in each pass.

The results show that using HLI can reduce the number of data dependences substantially. Most of the programs exhibited a reduction of over 60% in the number of dependence that must be assumed when using HLI compared to GCC's standard analysis. Floating-point programs obtain more reductions than integer programs on average.

Table 2 shows that for both integer and floating-point programs, over half of the data dependence

Benchmark	Total # of queries	# of queries in *Sched*	# of queries in *Loop*	# of queries in *CSE*	# of queries in *Local*
wc	950	212	152	543	43
023.eqntott	2200	398	348	1349	105
129.compress	1529	280	56	1177	16
008.espresso	21821	3827	5710	11738	546
Average	6625	904	1566	3702	180
101.tomcatv	1660	260	930	433	37
102.swim	3474	526	1728	1115	105
107.mgrid	3446	531	1905	970	40
103.su2cor	20697	3744	6271	10475	207
125.turb3d	64713	15152	799	39547	9215
034.mdljdp2	10094	2660	4022	2866	546
056.ear	3605	973	441	1981	210
052.alvinn	316	35	151	127	3
Average	27001	5970	4062	14379	2591

Table 2: The total number of data dependence queries for all four of the GCC optimization passes.

queries are in the *CSE* pass. The register local-allocation pass has the fewest dependence queries among the four passes tested. The number of queries in both *Sched* and *Loop* is between these two. The differences in the number of dependences for each optimization pass produces the different performance improvements for each pass, as discussed in Section 4.3. 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 *HLI_only results* and the *Combined results* in Figure 7 are often not the same. This difference means that there is room for additional improvement in the HLI. Current shortcomings in generating the HLI include: (1) the front-end algorithms for array data dependence and pointer analysis in SUIF are not as aggressive as possible, and (2) there are miscellaneous GCC code generation rules that the current HLI implementation has not considered. Ignoring these rules produces *unknown* dependence types between some memory references. The values of the *HLI result* are expected to become smaller as more aggressive front-end algorithms are developed and the current implementation limitations are overcome.

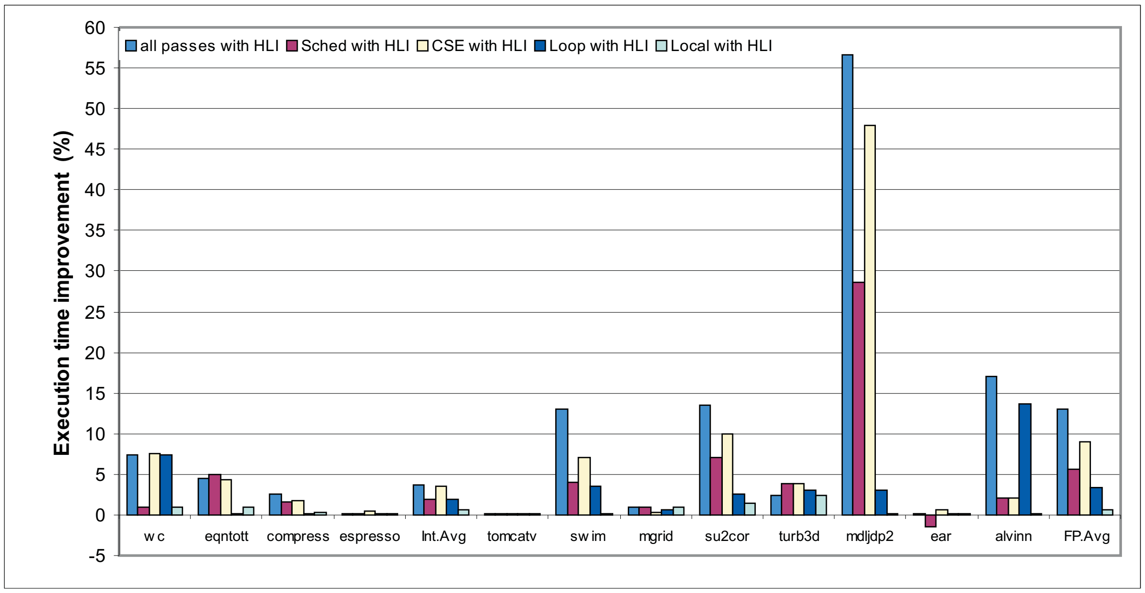


Figure 8: Impact on execution times when using HLI in the different GCC optimization passes for the MIPS R10000. (The bars for each program report the execution time improvement of using the HLI in all 4 optimization passes, in the *Sched* pass only, in the *CSE* pass only, in the *Loop* pass only and in the *Local* pass only over the execution time without using the HLI in any of the optimization passes. The solid horizontal lines for some programs are used to show that the execution time improvement is zero.)

4.3 Impact on program execution times

To study the performance improvement attributable to using HLI in GCC's four optimization passes, the benchmark programs were compiled in six different ways: without HLI, using HLI in all four optmization passes (All), using HLI in the *Sched* pass only (Sched), using HLI in the *CSE* pass only (CSE), using HLI in the *Loop* pass only (Loop), and using HLI in the *Local* pass only (Local). Execution times of these six different cases for each benchmark program were measured on a MIPS R10000 superscalar processor with 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* was 73 MB of C source code.

Figure 8 shows that, among the twelve programs tested, seven achieved a noticeable speedup in more than three of the cases. One program, *034.mdljdp2*, obtained remarkable speedups for the Sched, CSE and All cases. For almost all programs, the executables compiled with HLI used in all four passes achieved the best speedup due to benefits contributed by each pass.

The specific pass that obtains the most performance improvement depends on the characteristics of the program. Comparing Figures 7 and 8, we see that the reductions in the number of dependence arcs that come from using HLI tend to correlate with improvements in the execution time. However, a large reduction in the number of dependence edges does not always result in a correspondingly large execution time speedup, as can be seen in *125.turb3d*, for instance. One reason for this lack of improvement in execution time is because the optimiztion passes studied are limited to optimizations only within basic blocks. Another reason is that in some of the programs, the size of the most frequently executed basic blocks is relatively small. As a result, there are fewer memory references available to use the HLI during runtime. In addition, our pointer analyzer implemented on the SUIF front end is still preliminary and often makes conservative assumptions as it performs inter procedural pointer analysis. This makes the HLI information less aggressive in handling some pointer dereferences in a number of important functions in the benchmark programs. On the other hand, *034.mdljdp2* achieved the best performance improvement. The most frequently called functions in this program have a large number of memory references in very large basic blocks.

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 small, containing only 5 – 6 instructions on average. Furthermore, it is likely that each basic block contains few memory references. This is indirectly evidenced by comparing the total number of dependence queries made in the different programs tested (Table 2). Typically, an integer program requires fewer than one fourth the number of dependence tests needed by a floating-point program.

To summarize, using the HLI in the four optimization passes tested reduced the number of dependence edges by over 60% in the programs tested. This reduction translated into a moderate improvement in execution time. Expanding the scope of these optimizations beyond basic blocks can be enhanced by using the HLI. This expansion should lead to a more substantial reduction in execution times.

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 to identify parallel operations. Based on the results, a sequential program is transformed into a parallel program containing program constructs such as DOALL. Alternatively, the compiler may insert a directive before a sequential loop to indicate that the loop can be executed in parallel. Several research parallelizing Fortran compilers, including Parafrase [17], PFC [1], Parafrase-2 [24], Polaris [3], Panorama [11], and PTRAN [25], and commercial Fortran compilers, such as KAP [16] and VAST [32], 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. Once the thread assignment to individual processors has been determined, parallelizing compilers have little control over the execution of the code by each processor.

Over the past years, both *machine independent* and *machine specific* compiler techniques have been developed to enhance the performance of uniprocessors [6, 7, 15, 20, 21]. These compiler techniques rely primarily 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 publicly available GCC [28] and LCC [9] compilers exemplify the situation. They both maintain low-level IRs of the input programs, keeping no high-level program constructs for array data dependence and pointer-structure analysis.

With the increased demand for ILP, 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 pointer arguments [8, 34]. 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 data dependence analysis in a common uniprocessor compiler.

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 [26] 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 [33], 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 optimization 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.

6 Conclusions and Future Work

Instead of integrating the front-end and back-end into a single compiler, this paper proposes an 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 and back-end compilers.

We have demonstrated the effectiveness of this approach by implementing it within the SUIF front-end and the GCC back-end compilers. Our experiments with the SPEC benchmarks show that using this information in four optimization passes of GCC substantially reduces the number of data dependences compared with using standard GCC dependence analysis algorithm only. The increased flexibility provided by this reduction allowed GCC to improve execution time compared to using only the low-level information normally available in GCC.

We expect that the HLI mechanism proposed in this paper will make 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[3] that will use the HLI mechanism

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

to export high-level program information to the same GCC back-end implementation used in these experiments. The HLI will be used more extensively in back-end optimizations besides those done within basic blocks. We believe that global optimizations will provide HLI more potential to improve the performance of applications. Furthermore, compilers for future wide-issue processor architectures, such as *the Multiscalar architecture* [27], *the Superthreading architecture* [30] and *the Trace processor* [22], may benefit substantially from HLI when generating highly optimized codes to exploit the available hardware parallelism.

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. Sangyeun Cho was supported in part by a fellowship from the Korea Foundation for Advanced Studies.

References

[1] J. R. Allen and K. Kennedy. "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Trans. on Programming Language and Systems*, 9(4): 491 – 542, Oct. 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," *TR 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," *IEEE Computer*, pp. 78 – 82, Dec. 1996.

[4] S. Cho, J.-Y. Tsai, Y. Song, B. Zheng, S. J. Schwinn, X. Wang, Q. Zhao, Z. Li, D. J. Lilja, and P.-C. Yew. "High-Level Information – An Approach for Integrating Front-End and Back-End Compilers," *Proc. of the 1998 Int'l Conf. on Parallel Processing*, pp. 346 – 355, Aug. 1998.

[5] S. Cho and Y. Song. "The HLI Implementor's Guide (v0.1)," *Agassiz Project Internal Document*, Sept. 1997.

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

[7] J. C. Dehnert and R. A. Towle. "Compiling for the Cydra 5," *J. 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," *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pp. 242 – 256, June 1994.

[9] C. Fraser and D. 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," *Proc. of the 6th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, June 1997.

[12] L. Gwennap. "Intel's P6 Users Decoupled Superscalar Design," *Microprocessor Report*, Vol. 9, No. 2, Feb. 1995.

[13] L. Gwennap. "Digital 21264 Sets New Standard," *Microprocessor Report*, Vol. 10, No. 14, Oct. 1996.

[14] J. Heinrich. *MIPS R10000 Microprocessor User's Manual, V1.1*, MIPS Technologies, Inc., 1995.

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

[16] KAP User's Guide, *Tech. Report (Doc. No. 8811002)*, Kuck & Associates, Inc.

[17] D. J. Kuck *et al.* "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proc. of the 4th Int'l Computer Software and Application Conf.*, pp. 709 – 715, Oct. 1980.

[18] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, June 1988.

[19] M. H. Lipasti and J. P. Shen. "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE Computer*, pp. 59 – 66, Sept. 1997.

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

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

[22] E. Rotenberg, Q. Jacobson, Y. Sazeides, J. Smith. "Trace Processors," *Proc. of the 30th Annual Int'l Symp. on Microarchitecture*, pp. 138 – 148, Dec. 1997.

[23] Y. N. Patt, S. J. Patel, D. H. Friendly, and J. Stark. "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computer*, pp. 51 – 57, Sept. 1997.

[24] C. D. Polychronopoulos *et al.* "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," *Proc. of the 1989 Int'l Conf. on Parallel Processing*, Aug. 1989.

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

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

[27] G. S. Sohi, S. E. Breach, T. N. Vijaykumar. "Multiscalar Processors," *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture*, pp. 414 – 425, June 1995.

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

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

[30] J.-Y. Tsai and P.-C. Yew. “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation,” *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 35 – 46, Oct. 1996.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism,” *Proc. of the 22rd Annual Int'l Symp. on Computer Architecture*, pp. 392 – 403, May 1995.

[32] VAST-2 for XL FORTRAN, User's Guide, Edition 1.2, *Tech. Report (Doc. No. VA061)*, Pacific-Sierra Research Co., 1994.

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

[34] R. P. Wilson and M. S. Lam. “Efficient Context-Sensitive Pointer Analysis for C Programs,” *Proc. of the ACM SIGPLAN ‘95 Conf. on Programming Language Design and Implementation*, pp. 1 – 12, June 1995.