Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures*

Laurie J. Hendren and Guang R. Gao
School of Computer Science, McGill University
Montreal, Canada

Abstract

In this paper we propose a programming language mechanism and associated compiler techniques which significantly enhance the analyzability of pointer-based data structures frequently used in non-scientific programs. Our approach is based on exploiting two important properties of pointer data structures: *structural inductivity* and *speculative traversability*. Structural inductivity facilitates the application of a static interference analysis method for such pointer data structures based on *path matrices*, and speculative traversability is utilized by a novel loop unrolling technique for while loops that exploit fine-grain parallelism by speculatively traversing such data structures. The effectiveness of this approach is demonstrated by applying it to a collection of loops found in typical non-scientific C programs.

1 Introduction

In the past decade, the dramatic improvement of VLSI technology has led to modern high-performance microprocessors that support some level of fine-grain parallelism. Even with today's RISC processors, some degree of instruction-level parallelism is required to fully utilize the architecture. With ULSI (ultra-large scale integration) based multi-million transistor processor chips under way within this decade, we can expect an even higher degree of fine-grain parallelism due to deep instruction pipelining and multiple functional units (particularly multiple pipelined functional units). This rapid advance of VLSI technology and computer architectures has provided important challenges for programming language designers and compiler writers alike. It is becoming increasingly important to provide programming languages and associated compiler support such that user programs can be effectively *analyzed* in order to effectively utilize the underlying architectures. In particular, it is critical to provide compile-time analysis that results in accurate alias analysis and data-dependency information for complex data structures. In this paper we argue that such *analyzability* is an important principle of program language design and implementation, and that it is particularly critical for the efficient mapping of non-scientific programs to architectures supporting some level of instruction-level parallelism.

The importance of the analyzability has been demonstrated by the considerable success of automatic parallelization and optimization of large-scale scientific numerical programs. In such scientific programs, arrays are the most important data structures, and programs using these array structures can often be analyzed effectively. The keys to such analysis are: (1) arrays are frequently defined on rectangular index regions with dimensions, shapes, and bounds known to the compiler; (2) array operations are often encapsulated in "well structured" loops (**for** loops without **goto**s) with the iteration space of the loop matching the index regions of the arrays; and (3) in loops arrays are frequently accessed ("traversed") in a regular fashion. For example, the index expression is often an affine function of the loop indices. This mathematical structure of the arrays and the regularity of their accesses in embedded loops has lead to the development of a variety of *dependence analysis*, *loop transformation*, and *parallelization* techniques [1, 2, 3, 12, 16]. Although many of these techniques were pioneered in the areas of vectorizing and parallelizing compilers, these techniques are also being applied to architectures supporting instruction-level parallelism. Unfortunately, the analysis and optimization of non-scientific programs has not been so successful. In fact, some researchers have reported pessimistic results indicating that the exploitable instruction-level parallelism in non-scientific programs is so low as to question the merit of advanced advanced processor architectures such as superscalar/VLIW machines and superpipelined machines [9, 13, 14]. However, a

*This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

fresh look at the study by Wall actually shows impressive instruction-level parallelism assuming perfect alias analysis for all types of data structures [7, 14].

Given the increasing importance of fine-grain parallelism and the evidence that good alias analysis aids in the exploitation of such parallelism, it is clear that we must have programs that are analyzable. To illustrate both the importance of *analyzability* for programs with pointer data structures and the solution methodology proposed in this paper, we propose a programming language mechanism which significantly enhances the analyzability of pointer-based data structures frequently used in non-scientific programs. Our approach is based on two important properties : *structural inductivity* and *speculative traversability*. Structural inductivity facilitates the application of a static interference analysis method for such pointer data structures based on *path matrices*, and speculative traversability is utilized by a novel loop unrolling technique for while loops that exploit fine-grain parallelism by aggressively traversing such data structures. The effectiveness of this approach is demonstrated by applying it to a collection of loops found in typical non-scientific C programs.

The paper is organized as follows. In section 2 we present a more detailed outline of the challenges of compiling imperative programs with pointer data structures for machines with instruction-level parallelism. In section 3, we introduce programming language constructs which enhance the analyzability of programs by providing the programmer with a means of specifying two important properties of pointer data structures frequently found in non-scientific programs: *speculative traversability* and *structural inductivity*. In section 4 we give a concrete case study to show how such mechanisms can be effectively applied to unroll while loops that traverse such data structures. Lastly, we provide our conclusions in section 5.

2 Instruction-level parallelism and pointer data structures

In this section we outline the challenges of compiling for instruction-level parallelism in the presence of pointer data structures. In the first subsection we show how inaccurate alias analysis for pointer structures can greatly reduce the effectiveness of instruction scheduling and in the the second subsection we discuss the difficulties encountered in applying loop transformations in the presence of pointer data structures.

2.1 Instruction scheduling

In compiling for architectures with instruction-level parallelism, instruction scheduling is a crucial component. That is, in order to effectively utilize the architecture, the compiler must take a sequential list of instructions, and rearrange the instructions in such a way as to increase parallelism (reduce execution time) while preserving the original meaning of the program. Instruction scheduling is commonly performed for pipelined RISC architectures [10]. In order to fully utilize such instruction scheduling techniques, it is necessary to have accurate dependency analysis. Consider the small example program given in Figure 1(a). Using a naive translation for an architecture which has a 2 cycle latency for loads and additions, we get the 12 instructions shown in Figure 1(b)[1]. Due to the 2 cycle latency of load and addition operations, we had to insert `nop` instructions. For example, the `nop` between instructions (2) and (3) was inserted because we needed an extra cycle to wait for the load to `r2` to complete before we could compute `r1 + r2`. We can improve upon this schedule by noting that the 8 numbered instructions do not necessarily need to be executed in the order that they were generated. For example, if `0[a]` and `b` refer to different memory locations, then the load of `0[a]` does not depend on (does not need to proceed) the load of `b`. Thus, if we have a compile-time analysis that can determine that `0[a]`, `b`, `0[c]`, and `d` are all distinct variables (they all refer to different memory locations), then we can represent the partial order with the dependency diagram given in Figure 1(c). Starting from this partial order we can produce a much better schedule (8 cycles instead of 12 cycles) as shown in Figure 1(d). This improved schedule requires no extra `nop` instructions since all load and add instructions are scheduled at least two cycles before their results are needed. However, most compile-time analysis methods cannot determine whether or not the pointer variables `a` and `c` refer to the same memory location, and the dependencies will therefore be as shown in Figure 1(e). Using these dependencies we get the substantially longer schedule shown in Figure 1(f). Thus, it is clear that accurate alias analysis for pointer variables can lead to better instruction scheduling.

2.2 Loop transformations

There has been considerable effort spent on developing compile-time techniques that transform scientific programs in such a way as to expose more parallelism. In order to introduce the notion and benefit of loop transformations, let us consider *loop unrolling*, a

[1]Note that we use the notation `0[a] <-- r1` to indicate storing r1 into the memory location pointed to by `a`, and `r1 <-- 0[a]` to indicate loading r1 from the memory location pointed to by `a`.

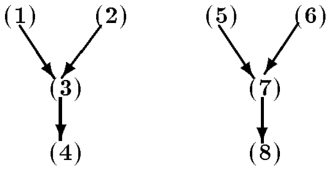
```
a->val = a->val + b;

c->val = c->val + d;
```

(a) C Program

```
(1)  r1 <-- 0[a]
(2)  r2 <-- b
     nop
(3)  r2 <-- r1 + r2
     nop
(4)  0[a] <--  r2
(5)  r3 <-- 0[c]
(6)  r4 <-- d
     nop
(7)  r4 <-- r3 + r4
     nop
(8) 0[c] <-- r4
```

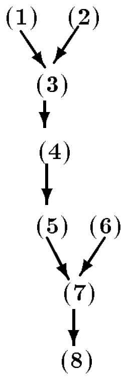
(b) Naive RISC schedule



(c) Good dependence analysis

```
(1)  r1 <-- 0[a]
(5)  r3 <-- 0[c]
(2)  r2 <-- b
(6)  r4 <-- d
(3)  r2 <-- r1 + r2
(7)  r4 <-- r3 + r4
(4)  0[a] <--  r2
(8)  0[c] <-- r4
```

(d) Good RISC schedule



(e) Bad dependence analysis

```
(1)  r1 <-- 0[a]
(2)  r2 <-- b
(6)  r4 <-- d
(3)  r2 <-- r1 + r2
     nop
(4)  0[a] <--  r2
(5)  r3 <-- 0[c]
     nop
(7)  r4 <-- r3 + r4
     nop
(8)  0[c] <-- r4
```

(f) Bad RISC schedule

Figure 1: Instruction scheduling example

transformation technique that was developed for parallelizing and optimizing compilers for scientific programs and arrays [4]. For example, loop unrolling can be used to transform the loop

```
for (i = 1; i <= 120 ; i++)
  a[i] = a[i] * b + c;
```

into the following equivalent unrolled loop that has multiple copies of the loop body.

```
for (i = 1; i <= 120; i = i + 3)
  { a[i] = a[i] * b + c;
    a[i+1] = a[i+1] * b + c;
    a[i+2] = a[i+2] * b + c;
  }
```

Loop unrolling was initially developed as a technique for reducing loop overhead and for exposing instruction-level parallelism for machines with multiple functional units. More recently it has also been applied in conjunction with instruction scheduling for pipelined and RISC architectures [15]. By increasing the size the body of the loop, the instruction scheduler can often produce a shorter schedule for the unrolled loop.

Now let us consider the problem of performing a similar transformation on while loops with pointer data structures. In Figure 2, we give three program fragments that were extracted from the source code for the GNU C compiler[2].

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```

(a) Initialize each item

```
prev = LIST_NULL;
while (lp != LIST_NULL)
  { next_lp = lp-> next;
    lp->next = prev;
    prev = lp;
    lp = next_lp;
  }
lp = prev;
```

(b) Reverse the list

```
if (lp != LIST_NULL)
  { next_lp = lp->next;
    while (next_lp != LIST_NULL)
      { lp = next_lp;
        next_lp = next_lp->next;
      }
  }
last = lp;
```

(c) Find last item

Figure 2: Three typical loops

The first loop, *initialize*, simply traverses a list initializing each key field to y. The second loop, *reverse*, destructively reverses the list. The third loop, *last*, traverses a list to find the last item. At first glance it would appear that the loops in Figure 2 are not really suited to techniques like loop unrolling. The potential problems include:

- In the case of arrays one can compute the index of array elements and reference many elements in parallel (`a[i], a[i+1], a[i+2]`). However, the pointer loops appear to be inherently sequential, and the list must processed by traversing each element in turn.
- In the array example it was straightforward to compute the loop bounds and increment for the unrolled loop. In the case of pointer loops we have no idea of the length of the structure, and no simple way to construct a termination condition that results in an equivalent unrolled loop.
- Even if we could access more than one item at a time, it would appear that loops like *last* have no com-

[2]We have modified the presentation of these loops so that they all use the same form and variable names.

putation to do on each item, and therefore may not be good candidates for loop unrolling.

• In the case of arrays the loop body will only modify the values within the structure, but not the structure itself. That is, the shape and size of the arrays remain the same. In the case of linked structures the loop may actually change the structure of the list itself. This is true in the case of *reverse*.

3 Analyzable pointer data structures

As illustrated in the previous section, there are many challenges for effectively compiling programs for instruction-level parallelism. In particular, we illustrated the negative effect of poor alias analysis for pointer data structures, and the difficulties encountered in designing loop transformations for pointer data structures. These difficulties arise because unlike data structures such as scalars and arrays which have a fixed shape and size, pointer data structures have dynamically changing shapes and sizes. In addition, programmers use dynamic data structures to implement a wide variety of structures including singly-linked lists, doubly-linked lists, circular lists, nested lists, binary trees, threaded trees, and graphs. Even though the programmer may use pointers in a very constrained manner (for example, the programmer may use a certain type of node to build only non-cyclical lists), the compiler has no way of knowing what sort of structure the programmer has in mind, and therefore the compiler cannot exploit any properties that are specific to that structure. In addition, the compiler has no knowledge of the size or length of a dynamic data structure.

In this section we introduce two characteristics of pointer data structures that allow better compile-time analysis. That is, we give some examples of how to make a programming language more *analyzable*. In the first sub-section we introduce the class of *structurally inductive* data structures and we illustrate how we can use *path matrix analysis* to provide accurate alias analysis for this class of data structures. In the second sub-section we introduce *speculative traversability*, a property that allows us to perform loop transformations without knowing the length of a data structure.

For practical reasons, including the wide-spread use of C for non-scientific programming, we have chosen to develop our approach relative to the programming language C. In fact, our techniques require only very small syntactic extensions to C that could be implemented with a straight-forward preprocessing step. However, one should not assume that the notion of designing analyzable programming languages is restricted to C, and we hope that others will take the challenge of designing better, analyzable programming languages that are suitable for architectures supporting instruction-level parallelism.

3.1 Structurally inductive pointer data structures and alias analysis

In order to provide more accurate alias analysis for pointer data structures, we would like programmers to be able to classify their data structures as either *inductive* or *non-inductive*. An *inductive* linked structure is one in which there are no cycles, and each node has at most one parent. *Inductive* structures include linked lists, nested lists, and trees, while *non-inductive* structures include DAGs and cyclic graphs. Inductive structures have nice properties for analysis, and techniques for alias analysis, dependence analysis, and parallelizing transformations for this class of structures have been developed [5, 8, 11]. The exploitable properties of *inductive* structures include: (1) the component pieces of the structure (the head and tail of a list, or the left and right sub-trees of a binary tree) never share any storage, and thus computations on the components are non-interfering, (2) breaking any link yields two independent pieces, and (3) a traversal of any series of linked nodes never revisits the same node more than once.

Our approach to exploiting the analyzability of inductive structures is to allow the programmer to indicate which pointer data structures have this property. In order to discuss our language extension with respect to C, we introduce a high-level recursive type statement, `rectype`. Each rectype statement consists a list of field names, with each field having either a scalar type, or a recursive reference to the type being defined. As illustrated by the example for linked lists as given in Figure 3, the `rectype` statement can be translated by the compiler into the traditional recursively defined pointer structures in C. Note that the compiler can generate both the appropriate C types for nodes and pointers to the nodes, and constant representing the empty structure (`LIST_NULL`).

As illustrated in Figure 3(a), we provide the option of indicating if a particular `rectype` has the *inductive* property. It should be noted that both *inductive* and *non-inductive* recursive structures are implemented with the same C data types. Thus, we treat the inductive specification as a directive, rather than a type. The programming language designer has a choice of how to use this directive. The easiest, and **least** safe, approach is to take this directive as a promise from the programmer that all structures built with this type will be inductive. This is analogous to allowing arrays without bounds checking. Another,

```
                   /* name of
                      the empty list */
rectype            #DEFINE LIST_NULL 0
LIST [inductive]
{ int key;         /* node type for
  LIST next;              ordinary lists */
}                  typedef struct LIST
                    { int key;
                      struct LIST *next;
                    } LIST_NODE, *LIST_PTR;
```

(a) *(b)*

Figure 3: A `rectype` and its C implementation

and perhaps preferable, approach is to use the directive to aid the compiler in choosing the correct sort of alias analysis to perform for that data type. Thus, an analysis best suited to inductive structures can be used for each `rectype` which has been labeled as inductive. Just as dependence analysis is a natural choice for analyzing array references, *path matrix* analysis is a natural choice for inductive structures [8].

Path matrix analysis is an interprocedural analysis technique that was specifically developed for inductive data structures. The technique exploits the special properties of inductive structures in order to provide accurate static analyses to: (1) safely determine if pointer data structures are guaranteed to be *inductive*, (2) perform alias analysis and dependency analysis for all pointer variables that point to nodes of the inductive structure, and (3) detect non-interfering computations.

We illustrate the use of path matrix analysis for the program fragment presented in Figure 4. This program first calls a function to build a list, and then executes a `while` loop to reverse the list. At the beginning of the loop, `orig_lp` points to the first item, while at the end of the loop, `new_lp` points to the first item of the reversed list, and `orig_lp` points to the last cell of the reversed list.

In order to demonstrate how the analysis works, we used our path matrix analysis tool[3] to process the program given in Figure 4 and we extracted pieces from the trace of the analysis as given in Figure 5. For each program point in the trace, we give the path matrix that summarizes the relationships (or paths) that exist between each pair of pointer variables (also called *handles*) live at that point. Also, note that each handle is decorated with a pair that looks something like $(\bullet,\odot)$. The first item of the pair corresponds to "nilness" of the handle itself, and the second item corresponds to the "nilness" of the *next* field of the handle ($\bullet$ means definitely not nil, $\circ$ means definitely nil, and $\odot$ means

[3] This is an interprocedural analysis tool that is written in SML [6, 8].

```
 /* build a list with n items */
 orig_lp = build_list(n);
/* orig_lp points to original head of list */
 lp = orig_lp;
 prev = LIST_NULL;
 while (lp != LIST_NULL)
   { /* get next node */
     next_lp = lp->next;
     /* reverse link of lp */
     lp->next = prev;
     /* get ready for next iteration */
     prev = lp;
     lp = next_lp;
   }
 /* new_lp points to new head of list, and
        orig_lp now points to last element */
 new_lp = prev;
```

Figure 4: An example program with the *reverse* loop

that it could be either). Let us consider the following points in the path matrix analysis.

Program Point (1): The first statement in our program fragment is a call to a procedure `build_list` that builds a non-cyclical list. Although we have not shown the path matrix analysis for this call, we see that the resulting path matrix contains only one live handle, *orig_lp*, and it has the relationship S with itself. Note that the relationship S between two handles x and y indicates that these variables point to the *same* node. The fact that there is only one entry in the path matrix indicates that there are no other live pointer variables (handles) that can be reached from the head of the list *orig_lp*, and that the analysis of the procedure call successfully determined that the structure pointed to by *orig_lp* is indeed inductive. If the analysis of the call had not been successful (for example, the programmer may have created a cyclic structure), then a less accurate alias analysis mechanism must be used.

Program Point(2): The path matrix computed for program point 2 contains two handles, *lp* and *orig_lp*. Note that the entries reflect the fact that *lp* and *orig_lp* point to the same node.

Program Point(3): In statement 3 we see the first occurrence of the nil pointer (`LIST_NULL`). You can see that the handle *prev* is definitely nil, and it is not related to any other handle.

Program Points(4) to (7): We have given the path matrices computed for the last iteration of the while loop fixed-point calculation. You can think of these path matrices as approximating the state of the data structures for all iterations of the while loop after the first iteration. Note that at program point (4) the relationship between *lp* and *next_lp* is always N^1 (exactly one *next* link), while the relationship between *prev* and *orig_lp* is $(S + N^+)$ (they point to the same node, or there is a chain of one or more *nexts* from *prev* to *orig_lp*). Also note that there is *no* relationship between *prev* and *lp*. This corresponds to the

`(1) orig_lp:=build_list(n)`

	orig_lp($\bullet$,$\odot$)
orig_lp	S

`(2) lp = orig_lp`

	orig_lp($\bullet$,$\odot$)	*lp*($\bullet$,$\odot$)
orig_lp	S	S
lp	S	S

`(3) prev = LIST_NULL`

	orig_lp($\bullet$,$\odot$)	*lp*($\bullet$,$\odot$)	*prev*($\circ$,$\circ$)
orig_lp	S	S	
lp	S	S	
prev			S

```
========= LAST ITERATION OF WHILE LOOP
          FIXED-POINT CALCULATION  ===========
(4) next_lp = lp->next
```

	orig_lp ($\bullet$,$\circ$)	*lp* ($\bullet$,$\odot$)	*prev* ($\bullet$,$\odot$)	*next_lp* ($\odot$,$\odot$)
orig_lp	S		$S?$	
lp		S		N^1
prev	$(S+N^+)$		S	
next_lp				S

`(5) lp->next = prev`

	orig_lp($\bullet$,$\circ$)	*lp*($\bullet$,$\bullet$)	*next_lp*($\odot$,$\odot$)
orig_lp	S		
lp	N^+	S	
next_lp			S

`(6) prev = lp`

	orig_lp($\bullet$,$\circ$)	*prev*($\bullet$,$\bullet$)	*next_lp*($\odot$,$\odot$)
orig_lp	S		
prev	N^+	S	
next_lp			S

`(7) lp = next_lp`

	orig_lp($\bullet$,$\circ$)	*lp*($\odot$,$\odot$)	*prev*($\bullet$,$\bullet$)
orig_lp	S		
lp		S	
prev	N^+		S

```
=========== END  WHILE/REPEAT ================
(8) new_lp = prev
```

	orig_lp($\bullet$,$\odot$)	*new_lp*($\odot$,$\odot$)
orig_lp	S	$S?$
new_lp	$(S+N^+)$	S

Figure 5: Partial trace of the path matrix computations

fact that the original list is now split into two distinct pieces, the reversed part starting at *prev* and ending at *orig_lp*, and the unreversed part starting at *lp*.

Program Point(8): Finally, at program point (8), we see the relationship of $(S + N^+)$ between the new head of the list *new_lp* and the original head *orig_lp*. The S corresponds to the case when the original list only has only one item, and the N^+ corresponds to all cases for lists with more than one item.

This example shows that we can get very accurate alias analysis for inductive structures when we apply alias analysis techniques that have been carefully designed to take advantage of the special properties of such structures. Indeed, the analysis provides accurate information even though the structure is being destructively traversed. This illustrates the point that one must be able to capture special properties of data structures at the programming language level, so that the appropriate compiler analysis techniques can be developed and used for those structures.

3.2 Speculatively traversable pointer data structures

In addition to providing properties that lead to better alias analysis, we must also deal with the problem of not knowing the length or size of pointer data structures. In order to fully demonstrate this problem, let us return to the problem of unrolling `while` loops. Recall that `for` loops that operate on arrays can be easily unrolled by simply modifying the counter and termination conditions of the loop. However, with while loops on pointers, the situation is much more difficult. Consider for example the *initialize* loop given in figure 6(a).

In order to effect some sort of unrolling, we can try the brute-force approach of duplicating the body of the loop, along with the appropriate conditionals. Figure 6(b) gives an example of this approach for unrolling the *initialize* loop 3 times. Although clearly semantically equivalent to the original loop, this approach does not appear to improve the code. The loop overhead is not improved because we need to do just as many tests, and the instructions in the body remain sequential. In figure 6(c) we give a good strategy for unrolling this loop. In this case two extra list pointers, `lp2` and `lp3` are introduced to give three independent pointers into the list. As a result, the three statements updating the fields `lp->key`, `lp2->key` and `lp3->key` can be executed in parallel, and the number of tests is reduced to 1/3 of the original loop. However, this unrolled loop is not necessarily semantically equivalent to the original loop. The most blatant problem is

```
while (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```
(a) Original Loop

```
while (lp != LIST_NULL)
{ lp->key = y;
  lp = lp->next;
  if (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
    if (lp != LIST_NULL)
      { lp->key = y;
        lp = lp->next;
      }
  }
}
```
(b) Brute-force approach

```
lp2 = lp->next;
lp3 = lp2->next;
while
 (lp3 != LIST_NULL)
  { lp->key = y;
    lp2->key = y;
    lp3->key = y;
    lp  = lp3->next;
    lp2 = lp->next;
    lp3 = lp2->next;
  }
while
 (lp != LIST_NULL)
  { lp->key = y;
    lp = lp->next;
  }
```
(c) Good unrolling

Figure 6: Loop Unrolling for the *initialize* example

that we don't know how many more items will be in the list, and the speculative computation of `lp2` and `lp3` may cause run-time errors that would not occur in the original loop. This leads us to define the property of *speculative traversability*. The idea of a speculatively traversable structure is that traversing the empty structure yields the empty structure. Thus, without knowing the length of a list, we can safely traverse the next k items without causing a run-time error. More formally, we define the following important property.

Definition 3.1 *Let t be the type of a* speculatively traversable *pointer data structure with scalar fields s_1, s_2, $\ldots$, s_m, and recursive pointer fields r_1, r_2, $\ldots$, r_n. If p is the pointer representing the empty structure for type t, then for each r_i the following equality holds: p->$r_i = p$.*

Note that a *speculatively traversable* recursive data structure is really a different type than an *ordinary* recursive data type because we have changed the meaning of operations on the empty structure. Therefore, to capture this idea we define a new type statement, `specrectype` as illustrated in figure 7. Also note that the implementation of the `specrectype` definition is identical to that of the `rectype` except for the definition of the empty data structures. That is, we have implemented the empty structure so that we can speculatively traverse the data structure (extra traversals on the empty structure will always result in the empty structure).

4 Loop transformations

As we discussed in section 2.2, loop-based transformations are important components of compiling for

```
specrectype LIST
{ int key;
  LIST next;
}
```

```
/* name of the empty list */
#define LIST_NULL list_nil
/* node for implementing empty list */
static LIST_NODE list_nil_node;
/* pointer to the empty list */
static LIST_PTR list_nil;

/* code to initialize the empty list ... */
void init_LIST_NULL()
{ /*  set pointer to empty list node       */
  list_nil = &list_nil_node;
  /*  set each scalar type to approp. zero */
  list_nil->key = 0;
  /*  set each recursive field to self     */
  list_nil->next = list_nil;
}

/* node type for speculative list */
typedef struct LIST
 { int key;
   struct LIST *next;
 } LIST_NODE, *LIST_PTR;
```

Figure 7: A `specrectype` and its C implementation

instruction-level parallelism. In this section we present a new loop unrolling technique that applies to structurally inductive and speculatively traversable pointer data structures. In the second part of this section we provide experimental results that indicate substantial performance gain for our loop unrolling technique.

In order to demonstrate a wide variety of loops we consider the three loops already presented in Figure 2 and three additional loops presented in Figure 8. The *sum* loop is typical of a loop that is traversing a structure while accumulating a final value. The *count* loop is an example of a loop that performs some action on a subset of the items in the list. An important characteristic of this loop is that it contains a conditional in the body. The *find* loop is typical of a loop that does not traverse the entire list. The characteristic of importance in this loop is the more complex termination condition.

4.1 A new loop unrolling technique

Each of the loops given in Figure 8 traverses and processes a linked list one item at a time. That is, for each iteration of the while loop, one item of the list is processed. The basic strategy of our loop unrolling is to transform the original loop into a loop which processes more than one item on each iteration. The advantages of such a transformation include: (1) loop overhead is reduced, (2) the size of the loop body

```
sum = 0;
while (lp != ORIG_NULL)
  { sum += lp->key;
    lp = lp->next;
  }
```

Sum all elements

```
count=0;
while (lp != LIST_NULL)
  { if (lp->key == y)
        count++;
    lp = lp->next;
  }
```

Count equal to y

```
while ((lp != LIST_NULL) &&
                (lp->key != y))
    lp = lp->next;
```

Find first element equal to y

Figure 8: Some additional test loops

is increased, thereby providing more flexibility for traditional optimizations like dead code removal and instruction scheduling, and (3) in many cases available parallelism is increased because the operations on different list items may proceed independently.

We have isolated two general patterns for performing such while loop transformations. In Figure 9 we give the two patterns and the equivalent *3-unrolling* (there is an obvious generalization for *k-unrolling*). All of our example loops, except for *reverse*, fit pattern A. Pattern B is typical of loops, like *reverse*, that are updating the structure of the list as it is traversed. Note that in both cases, the unrolling of the pattern consists of two adjacent while loops. The first loop processes k items for each iteration, and the second loop processes the remaining tail of the list in the case that the length of the list is not a multiple of k.

The following is a concise summary of the loop unrolling method.

Detection: Each while loop is examined to see if it fits the pattern A or B. In either case, the loop must obey the following properties:

(1) The condition, *cond(lp)* must be some boolean expression defined on the variable *lp* and this variable must be of `specrectype`.

(2) The body of the loop must be divisible into two *non-interfering* sub-pieces, *body(lp)* and *traverse(lp)*. This analysis can be done by simple symbolic inspection for some loops, or it can require *path matrix* analysis for more complex loops like *reverse*. The *traverse(lp)* computation must be a traversal of a recursive field of the `specrectype`.

Unrolling: The loop is unrolled by introducing new variables *lp2, lp3, ..., lpk* of the appropriate `specrectype` and producing a pair of new loops as illustrated in Figure 9. The *k*th copy of the body is created by replacing each occurrence of *lp* with *lpk*.

Conditional optimization: The naive loop unrolling creates the conditional "*cond(lp) && cond(lp2) && ... && cond(lpk)*". This can often be greatly

```
while (cond(lp))
  { body(lp);
    lp = traverse(lp);
  }

lp2 = traverse(lp);
lp3 = traverse(lp2);
while (cond(lp,lp2,lp3))
  { body(lp);
    body(lp2);
    body(lp3);
    lp = traverse(lp3);
    lp2 = traverse(lp);
    lp3 = traverse(lp2);
  }
while (cond(lp))
  { body(lp);
    lp = traverse(lp);
  }
```

(a) Pattern A

```
while (cond(lp))
  { new_lp = traverse(lp);
    body(lp);
    lp = new_lp;
  }

lp2 = traverse(lp);
lp3 = traverse(lp2);
while (cond(lp,lp2,lp3))
  { new_lp = traverse(lp3);
    body(lp);
    body(lp2);
    body(lp3);
    lp = new_lp;
    lp2 = traverse(lp);
    lp3 = traverse(lp2);
  }
while (cond(lp))
  { new_lp = traverse(lp);
    body(lp);
    lp = new_lp;
  }
```

(b) Pattern B

Figure 9: Patterns for unrolling 3 times

simplified. For example, using property of speculatively traversable structures defined in section 3.1, we can simplify "*(lp != LIST_NULL) && (lp2 != LIST_NULL) && (lp3 != LIST_NULL)*" to "*(lp3 != LIST_NULL)*".

Loop body optimization: We note that in the unrolled loops there are many adjacent copies of the body of the original loop. This often provides further opportunities for traditional optimizations such as *copy elimination* and *dead code removal*. Figure 10(c) illustrates the use of *dead code removal* for the loop *last*.

Parallelization: As a final step we determine if the different copies of the body are independent. In determining this we can make use of the fact that a particular linked structure is inductive. For example, in the case of inductive structures, we can guarantee that the k different variables *lp, lp2, ..., lpk* refer to independent nodes.

4.2 Experimental results

In this section we present some experimental results obtained by applying our loop unrolling techniques to the six loops presented in Figures 2 and 8. First let us consider the example of unrolling the loop *last*. In Figure 10 we show the original loop, the unrolled loop, and the unrolled loop after both the conditional and body have been optimized. Note that these optimizations are straight-forward applications of the *speculatively traversable* property and *dead code elimination*. Due to the resulting reduced loop overhead and the removal of unneeded computation in the unrolled version of *last*, we would expect improved performance

```
next_lp2 =
    next_lp->next;
next_lp3 =
    next_lp2->next;
while (
(next_lp!=LIST_NULL)&&
(next_lp2!=LIST_NULL)&&
(next_lp3!=LIST_NULL))
{ lp = next_lp;
  lp = next_lp2;
  lp = next_lp3;
  next_lp =
      next_lp3->next;
  next_lp2 =
      next_lp->next;
  next_lp3 =
      next_lp2->next;
}
while
(next_lp != LIST_NULL)
{ lp = next_lp;
  next_lp =
      next_lp->next;
}
```

(a) Unrolled

```
next_lp2 =
    next_lp->next;
next_lp3 =
    next_lp2->next;
while
(next_lp3 != LIST_NULL)
{ lp = next_lp3;
  next_lp =
      next_lp3->next;
  next_lp2 =
      next_lp->next;
  next_lp3 =
      next_lp2->next;
}
while
 (next_lp != LIST_NULL)
{ lp = next_lp;
  next_lp =
      next_lp->next;
}
```

(b) Unrolled & optimized

Figure 10: Unrolling *last*

of the unrolled version over the original loop. This performance improvement is confirmed with the experimental Figures in table 1. This experiment was performed using the original loop, and optimized unrolled loops for k equal to 2, 3, 4, 5, and 10. In each case, the transformation was performed at the C source level, and then the resulting unrolled loop was compiled using the native `cc` compiler with the `-O` optimizer option. For each unrolling we give the time in microseconds required to execute the loop for lists of length 10, 100, and 1000. In addition, the speedup is indicated in parentheses. We note excellent speedups ranging from 1.12 to 2.21 for all cases except for the 10-unrolling run on a list with 10 elements. In this case, the cost of the speculative traversal outweighs the other benefits. Also, for both architectures, we note that a 3-unrolling results in very good performance for all lengths of lists (speedups ranging from 1.21 to 1.88).

We have also performed a complete set of experiments on each of the six loops presented in this paper. For each loop we experimented with the effect of the technique with a variety of C compilers, and for each C compiler the effect when used with or without the `-O` optimizing option. Table 2 summarizes part of those results for the case of 3-unrolling and using the native `cc` compiler with the `-O` option. This resulted in impressive speedups for all cases on the MIPS architecture (average speedups of 1.19 to 1.49), and good

SPARC[4]

N	*Orig Loop*	*2 unroll*	*3 unroll*	*4 unroll*	*10 unroll*
10	1.72 **(1.00)**	1.50 **(1.15)**	1.42 **(1.21)**	1.54 **(1.12)**	2.46 **(0.70)**
100	15.40 **(1.00)**	11.60 **(1.33)**	10.40 **(1.48)**	10.00 **(1.54)**	9.60 **(1.60)**
1000	152.00 **(1.00)**	102.00 **(1.49)**	102.00 **(1.49)**	104.00 **(1.46)**	88.00 **(1.73)**

MIPS[5]

N	*Orig Loop*	*2 unroll*	*3 unroll*	*4 unroll*	*10 unroll*
10	2.14 **(1.00)**	1.66 **(1.29)**	1.52 **(1.41)**	1.67 **(1.28)**	2.95 **(0.73)**
100	20.23 **(1.00)**	12.42 **(1.63)**	11.17 **(1.81)**	10.86 **(1.86)**	10.86 **(1.86)**
1000	202.33 **(1.00)**	121.87 **(1.66)**	107.81 **(1.88)**	102.34 **(1.98)**	91.40 **(2.21)**

Table 1: Speedups for *last*

speedups on most cases for the SPARC architecture (average speedups of 1.03 to 1.17). By studying the code produced by the various `cc` compilers, we note the following reasons for this speedup: (1) reduced number of branches, (2) reduced number of instructions, and (3) better instruction scheduling due to the increased size of the block body.

	SPARC		
	10	100	1000
last	1.42(**1.21**)	10.40(**1.48**)	102.00(**1.49**)
initialize	2.14(**1.00**)	21.00(**1.00**)	242.00(**0.96**)
reverse	2.40(**1.13**)	22.60(**1.06**)	212.00(**1.16**)
count	2.80(**1.06**)	23.20(**1.17**)	224.00(**1.21**)
sum	2.16(**0.99**)	17.20(**1.08**)	174.00(**1.03**)
find	1.78(**0.78**)	12.60(**1.21**)	122.00(**1.15**)
average	**(1.03)**	**(1.17)**	**(1.17)**

	MIPS		
	10	100	1000
last	1.52(**1.41**)	11.17(**1.81**)	107.81(**1.88**)
initialize	1.70(**1.26**)	13.83(**1.47**)	134.37(**1.51**)
reverse	1.77(**1.26**)	13.98(**1.45**)	136.71(**1.49**)
count	2.95(**1.14**)	27.50(**1.19**)	274.20(**1.20**)
sum	1.70(**1.26**)	13.75(**1.48**)	135.15(**1.51**)
find	1.91(**0.82**)	14.14(**1.15**)	135.15(**1.20**)
average	**(1.19)**	**(1.42)**	**(1.46)**

Table 2: Speedups for six benchmarks with an unrolling of 3

The observation that our loop unrolling technique lead to better instruction scheduling on RISC machines shows that techniques like this are very important for the effective exploitation of parallelism available in today's advanced architectures. Furthermore, when more refined alias analysis based on path matrix

[4] SPARCstation 2, Sun release 4.1 cc compiler

[5] DECSTATION 5000, MIPS cc compiler

analysis is incorporated one would expect even further speedups due to improved instruction scheduling.

5 Conclusions

Current and future generation of high-performance architectures support instruction-level parallelism. To produce efficient code for such machines, a compiler must be able to analyze programs and detect opportunities for optimization and parallelization. This motivates the central theme of this paper: analyzability is an important principle for programming language design and implementation.

In this paper we focused on the analyzability issues for real-life non-scientific programs. A majority of such programs are being written in imperative languages like C for many different hardware platforms, and there is no sign that this trend will slow down soon. Therefore, we feel that one challenge for researchers in the areas of programming language design and implementation is to work towards solutions that: (1) are easily assimilated and adapted by the community programming in C-like languages, and (2) will lead to effective use of current and future high-performance architectures.

We illustrated the negative effect of poor alias analysis for pointer data structures, and the difficulties encountered in designing loop transformations for them. These difficulties arise because unlike data structures such as scalars and arrays, pointer data structures have dynamically changing shapes and sizes. In addition, programmers use dynamic data structures to implement a wide variety of structures including singly-linked lists, doubly-linked lists, circular lists, nested lists, binary trees, threaded trees, and graphs.

We have proposed a programming language mechanism which can be utilized to design analyable pointer data structures with two important properties : *structural inductivity* and *speculative traversability*. We illustrate how can use *path matrix analysis* to provide accurate alias analysis for structural inductive data structures. We have also described a novel while loop transformation method to aggressively exploit fine-grain parallelism for pointer data structures which are speculatively traversable and we have provided experimental results that show that the transformation results in significant performance improvement.

References

[1] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. *Conference Record of the 14th POPL*, pages 63–76, January 1987.

[2] Todd Allen, Michael Burke, and Ron Cytron. A practical and powerful algorithm for subscript dependence testing. Technical report, IBM, 1986. Internal Report.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[4] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[5] Vincent A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[6] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[7] Laurie J. Hendren and Guang R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. ACAPS Technical Memo 28, McGill University, 1991.

[8] Laurie J. Hendren and Alex Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 1990.

[9] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the ASPLOS III*, pages 272–282, Boston, MA, 1988.

[10] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, 1990.

[11] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988*, pages 100–110, July 1988.

[12] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[13] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits of instruction issue. *Proceedings of ASPLOS III*, August 1989.

[14] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of ASPLOS IV*, pages 176–188, April 1991.

[15] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of ASPLOS II*, pages 105–109. ACM, 1987.

[16] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989.