To appear in Proceedings of the Twenty Fifth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, San Diego, California, January 19–21, 1998. ©1998 ACM (see notice below).

Putting Pointer Analysis to Work *

Rakesh Ghiya and Laurie J. Hendren
School of Computer Science, McGill University
Montréal, Québec, CANADA H3A 2A7
{ghiya,hendren}@cs.mcgill.ca
(514) 398-4657/398-7391

Abstract

This paper addresses the problem of how to apply pointer analysis to a wide variety of compiler applications. We are not presenting a new pointer analysis. Rather, we focus on putting two existing pointer analyses, points-to analysis and connection analysis, to work.

We demonstrate that the fundamental problem is that one must be able to compare the memory locations read/written via pointer indirections, at different program points, and one must also be able to summarize the effect of pointer references over regions in the program. It is straightforward to compute read/write sets for indirections involving stack-directed pointers using points-to information. However, for heap-directed pointers we show that one needs to introduce the notion of *anchor* handles into the connection analysis and then express read/write sets to the heap with respect to these anchor handles.

Based on the read/write sets we show how to extend traditional optimizations like common subexpression elimination, loop-invariant removal and location-invariant removal to include pointer references. We also demonstrate the use of our information on more advanced techniques such as array dependence testing and program understanding. We have implemented our techniques in our McCAT C compiler, and we demonstrate examples of applying our methods on a set of pointer-intensive C benchmarks, as well as present concrete empirical data on the improvements achieved.

1 Introduction and Motivation

Pointer analysis has recently been a subject of active research. This paper focuses not on a new pointer analysis, but rather on how the results of two existing pointer analyses can be used for a wide variety of compiler applications. That is, once the relationships between pointers are computed, what can we do with it? How do we put pointer analysis to work?

A variety of effective techniques have been proposed to estimate points-to or alias relationships for C [6, 10, 19, 26, 29, 31, 33, 35]. A common feature of all these techniques is that they approximate relationships between named objects. For objects that are on the stack, the appropriate variable names are used, while dynamically-allocated objects are handled by associating them to some set of static names.[1] This approach, where all memory locations are named, has several advantages. Firstly, the same analysis can be used for pointers to stack objects (*stack-directed pointers*) and pointers to heap objects (*heap-directed pointers*). Secondly, since names are static, it is quite simple to use the information in subsequent compiler analyses. However, treating the heap as a static set of named locations can also lead to significant imprecision [8], and substantial improvements in accuracy can be achieved when special *heap analyses* are used for heap-directed pointers [5, 9, 12, 13, 16, 27]. These approaches have focused on using different kinds of abstractions in order to get more precise or richer descriptions about the relationships between heap-directed pointers. However, a question remains about how to use the information provided by these analyses in subsequent compiler transformations.

Our approach has been to handle the stack and heap problems separately. We first resolve all pointer relationships on the stack using a *store-based* points-to analysis [10], which abstracts all heap locations as a single symbolic location called *heap*. All pointers reported to be pointing to *heap* are then further analyzed via a hierarchy of *storeless* heap analyses, *connection analysis* [12], and *shape analysis* [13]. The focus of this paper is to examine how the combination of points-to anal-

*This work supported by NSERC and FCAR.

[1] The simplest solution is to use only one name called `heap`, while more accurate solutions use some variant of `malloc` sites (i.e. associate each malloc site in the program with a name).

ysis and connection analysis can be used to compute information that can be used for a wide range of compiler applications. Connection analysis was chosen because it is a relatively simple type of *storeless* analysis that does not give names to all heap locations. Thus, it demonstrates the problems in using the results of a storeless analysis, and yields interesting results when "put to work" properly. The main contributions of this paper are as follows.

Computing read/write sets based on a storeless pointer analysis: Computing the set of locations read/written by a statement or program block is relatively simple when based on a store-based analysis [6, 20]. We also provide a brief description in Section 2.1. However, for a storeless analysis like connection analysis, a central problem is that even though one has fairly accurate information at each program point, one does not have static names for heap locations, and thus it is difficult to relate information known at one program point to information known at another program point. Further, it is not immediately obvious how to summarize the information for many program points (i.e. summarize the effect of a function body). Our solution is to create just enough names for heap objects, called *anchor handles*, so that we maintain the advantages of a storeless analysis, and at the same time we can use the information about these named anchor handles to relate different program points, and to summarize effects over many program points. We have implemented a connection analysis augmented with anchor handles, and a subsequent analysis that computes read/write sets relative to those handles.

Applications based on read/write sets: Based on read/write sets, we demonstrate how to use the information for a wide variety of applications including: (1) extending standard scalar compiler transformations, like loop-invariant removal, location-invariant removal, and common subexpression elimination, to include pointers references; (2) providing improved input to array dependence testers; and (3) providing summary information that is useful for program understanding, dynamic compilation [2] and prefetching of pointer data structures [24].

Implementations and Empirical studies: We have implemented our techniques in the McCAT compiler, and we present empirical data to illustrate the costs and benefits of the techniques. By performing source-to-source scalar transformations based on our read/write sets, we demonstrate up to 10% performance improvement over `gcc -O3`. For array dependence testers we show significant improvements with pointer read/write sets, and we demonstrate the use of our read/write sets for program understanding via a tool that produces output that can be browsed via Web browsers. Thus, we feel that we have demonstrated many practical applications of pointer analysis.

The paper is organized as follows. In Section 2, we present the necessary background for points-to analysis and connection analysis, and discuss how to compute read/write sets based on both analyses. In Section 3 we illustrate scalar optimizations and discuss other applications of read/write sets. We summarize related work in Section 4, and in Section 5 we draw conclusions and discuss future work.

2 Foundations

Our main goal is to identify the set of locations read/written by a given statement or program region. Consider the small program fragment in Figure 1. For statement `S` it is straightforward to compute that Read(S) = {`y, z`} and Write(S) = {`x`}. However, statements `T` and `U` involve indirect references and using the simple syntax-based approach above gives Read(T) = {`q, *q`}, Write(T) = {`p`}, and Read(U) = {`q, y`}, Write(U) = {`*q`}. This information is not sufficient to correctly identify interstatement dependencies. For example, Read(T) may conflict with Write(S), as the indirect reference `*q` can potentially access the same location as the variable `x`. Similarly, Write(U) may not conflict with Read(T), as the target of pointer `q` can change between the two statements. Thus in order to relate the read/write sets of statements involving indirection with those of other statements, one needs to resolve indirect references into a set of static locations. In the next two sections we outline the methods for accurately computing read/write sets using points-to analysis and connection analysis.

```
S: x = y + z;
T: p = *q;
 ...
U: *q = y;
```

Figure 1: Example for Read-Write Sets

2.1 Points-to Analysis and Read/Write Sets

Points-to analysis is a stored-based pointer analysis i.e. it specifically relies on the fact that all pointer targets have a compile-time name. It calculates pointer targets in terms of program point specific points-to triples of the form `(x,y,D)` or `(x,y,P)`. The triples respectively denote that the variable `x` definitely/possibly contains the address of the location corresponding to `y`. Symbolic names are generated for pointer targets outside the scope of the current procedure. Since heap locations are inherently anonymous, they are abstracted as one symbolic stack location called *heap*.

The detailed description of our context-sensitive interprocedural points-to analysis can be found in [10]. Here we simply illustrate it using the small code fragment shown in Figure 2. At program point `C` we have the points-to triples `(s, ptA, D)` and `(t, ptB, D)`,

```
typedef struct point
{ double x;
  double y;
  struct point *next;
}Point;
```

```
void flip(Point *a, Point *b)
/* a gets the reflection of b */
{ S: a->x = b->y;
  T: a->y = b->x;
}
void sum(Point *c, Point *d)
{ U: c->x = c->x + d->y;
  V: c->y = c->y + d->x;
}
```

```
void main()
{ Point *s, *t, ptA, ptB;
  int y;
    s = &ptA;
    t = &ptB;
 C: sum(s, t);
    s = alloc_point();
    t = alloc_point();
 D: flip(s, t);
 E: y = s->x;
}
```

Figure 2: Example for Connection-based Read-Write Sets

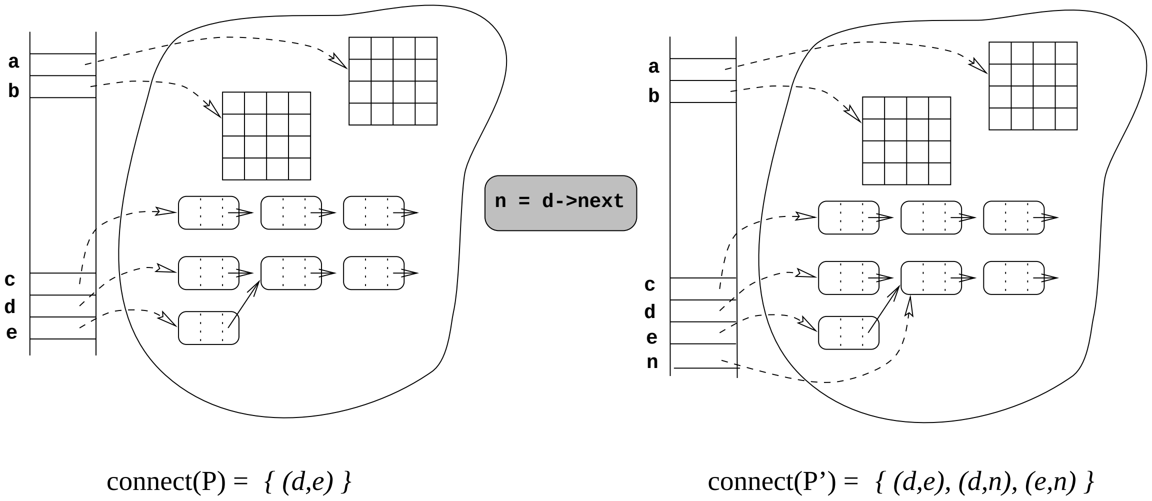


Figure 3: Snapshots and related connection information

which get respectively *mapped* as `(c, 1_c, D)` and `(d, 1_d, D)` at the entry of function `sum` (program point `U`). Note that `1_c` and `1_d` are symbolic locations generated to represent `ptA` and `ptB` (local to `main`) inside function `sum`. Based on this information, the read/write sets for statements `U` and `V`, and for the entire function `sum` can be easily computed as:

Read(U) = {`c, d, 1_c.x, 1_d.y`} Write(U) = {`1_c.x`}
Read(V) = {`c, d, 1_c.y, 1_d.x`} Write(V) = {`1_c.y`}
Read(sum) = {`c, d, 1_c.x, 1_c.y, 1_d.x, 1_d.y`}
Write(sum) = {`1_c.x, 1_c.y`}

From this information, interstatement dependencies for the function `sum` can be easily detected. Finally, the read/write sets for the function call to `sum` at program point `C` are computed by *unmapping* the read/write sets for the function `sum`, giving the following information:

Read(C) = {`s, t, ptA.x, ptA.y, ptB.x, ptB.y`}
Write(C) = {`ptA.x, ptA.y`}

Now consider the program point `D`. Here we have the points-to triples (`s`, *heap*, `P`) and (`t`, *heap*, `P`), which get mapped as (`a`, *heap*, `P`) and (`b`, *heap*, `P`) at the entry of function `flip`. With this information, we get very conservative read/write sets: Read(S) = Write(S) = Read(T) = Write(T) = {*heap*}, which indicate a false dependence between `S` and `T`. To obtain sharper read/write information for heap-directed pointers, we further perform a *heap analysis* called connection analysis, described below.

2.2 Connection Analysis and Read/Write Sets

Connection analysis is a *storeless* heap pointer analysis, i.e., instead of explicitly computing the potential targets of a pointer, it computes *connection* relationships between pointers. It is performed after points-to analysis, and focuses only on pointers reported to be heap-directed by points-to analysis. Two heap-directed pointers are *connected* if they *possibly* point to heap objects belonging to the same data structure. They are *not connected* if they *definitely* point to objects belonging to disjoint data structures. Figure 3 demonstrates a snapshot of a typical stack and heap at some program point P, and then another snapshot after executing the statement `n = d->next` (program point P'). Below the snapshots we give the connection relationships that hold at each point. First consider program point P. At this point there are five pointer variables that are heap-directed. Variables `a` and `b` point to disjoint arrays, and so they are not connected. Variables `c`, `d` and `e` point to various linked-list nodes. In this case only `d` and `e` point to the same data structure, and so they are the only variables that are connected. Thus, a valid connection set at P is $\{(d, e)\}$. After executing the statement `n = d->next`, `n` must be connected to all handles that were previously connected to `d`. Thus, a valid connection set at P' is $\{(d,e),(n,d),(n,e)\}$. Note that it is the negative information that is really useful. For example, the pair (a,b) is not in the set of connection relationships, so any operation on array `a` must be distinct from any operation on array `b`. Similarly any access to the list pointed to by `c` must be distinct from any access to the list pointed to by `d`. Also, note that connection relationships are given only in terms of program variables, and the actual objects in the heap are not given any names. Thus, even though the nodes of all the lists may have been allocated by the same static `malloc` site, connection analysis can determine that the lists are disjoint. Connection analysis is a context-sensitive interprocedural analysis, and its detailed description can be found in [12].

2.2.1 Difficulties in computing read/write sets based on connection analysis

Since connection analysis is storeless, it is difficult to relate connection information at different program points. Consider again the function `flip` in Figure 2. At call-site `D` of function `flip`, pointers `s` and `t` point to disjoint heap objects and are not connected. Consequently, at the entry to `flip` parameters `a` and `b` are not connected, and they remain disconnected over the entire function body (no statement connects them). Suppose we want to summarize the locations read and written relative to the parameters `a` and `b`. Based on the connection information we may deduce that the sets Write(S) = {`a->x`}, and Read(T) = {`b->x`} do not conflict, as `a` and `b` point to disjoint objects at both `S` and `T`.

However, such simplistic solutions do not suffice. Consider the function, `tricky_flip`, given in Figure 4(a). As illustrated in Figures 4(b) and 4(c), the pointer values stored in `a` and `b` change. Thus, even though `a` and `b` are not connected at point `S`, and also not connected at point `T`, there is a flow dependence since `a->x` at point `S` is the same location as `b->x` at point `T`.

2.2.2 Dropping Anchors

The fundamental problem in the `tricky_flip` example is that we don't have enough names for heap objects. The same programmer-defined name may refer to different objects at different program points. The solution is to invent enough new names, *anchor handles*, so that further analyses can be done simply, based on information relative to the anchor handles. Consider a new version of `tricky_flip` as given in Figure 5. In this example, we have introduced the names `a@tricky_flip` and `b@tricky_flip` which anchor the parameters, and we have introduced `a@S`, which is an anchor for variable `a` at program point `S`. Anchor handles are given their initial values by introducing *ghost copy statements*. All anchors for parameters are copied at the beginning of a function body, while program-point-specific anchors of the form `a@S` are copied at the given program point `S`.[2] Although we have shown these as part of the program, in fact, the anchors and ghosts are only implemented in the connection analysis, and the actual program is not modified.

Anchor handles serve as "anchor points" for analysis within the body of a function. Informally, we can now compute the read/write sets with respect to heap-related indirect references by noting that an anchor handle `x@p` is read/written each time any pointer connected to `x@p` is read/written. For example, as illustrated in Figure 5, at statement `S`, `a` is connected with anchors `a@tricky_flip` and `a@S`, and so we would indicate that writing `a->x` actually writes to the anchored locations `a@tricky_flip->x` and `a@S->x` giving HeapWrite(S) = {`a@tricky_flip->x, a@S->x`}. Similarly, at statement `T`, `b` is also connected with anchors `a@tricky_flip` and `a@S`, so the read `b->x` actually reads the anchored locations `a@tricky_flip->x` and `a@S->x` giving HeapRead(T) = {`a@tricky_flip->x, a@S->x`}. Comparing the sets HeapWrite(S) with HeapRead(T), one can detect the flow dependence from `S` to `T`.

We can also collect heap read/write sets with respect to function parameters in terms of their anchors. In our example, for the function `tricky_flip`, we have the following information:

HeapWrite(tricky_flip) = HeapRead(tricky_flip) =
{`a@tricky_flip->x, b@tricky_flip->y`}

This function level information could be used to determine that at the entry of function `tricky_flip`: (1) it might be useful to prefetch `a->x` and `b->y`, but one should not prefetch `b->x` and `a->y`; and (2) there are no updates to the `next` field of either `a` or `b`, and thus `tricky_flip` does not change the "listness" of any data structure.

2.2.3 Introducing Anchor Handles into Connection Analysis

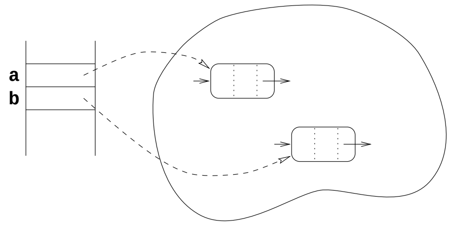
Introduction of too many anchor handles can affect the efficiency of connection analysis. Hence identifying the program points where anchor handles need to be introduced, and selecting the locations to be anchored, is an important issue.

In our implementation, for each function in the program, anchor handles are generated for each: (i) heap-directed formal parameter, (ii) heap-directed global pointer accessed in the function, (iii) call-site that can read/write a heap location, and (iv) heap-related indirect reference in the function body (`*p` is considered heap-related if the entry $(p, heap, P)$ is in the

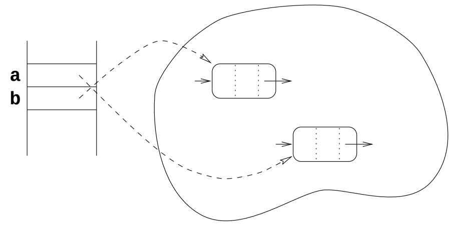
[2] For the sake of clarity we do not include other anchors like `b@S` and `a@T` in this example.

```
void tricky_flip(a,b)
Point *a; Point *b;
{ Point *tmp;

S: a->x = b->y;
   /* swap a and b */
   tmp = a; a = b;
   b = tmp;
T: a->y = b->x;
}
```

(a) **tricky_flip**



(b) program point S, connect(S) = {}

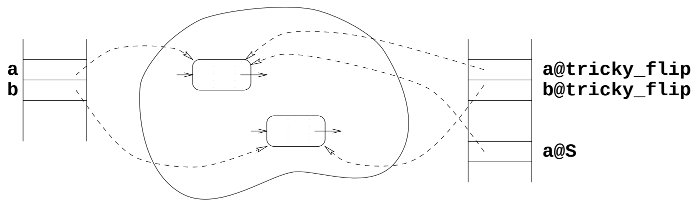


(c) program point T, connect(T) = {}

Figure 4: Tricky Example

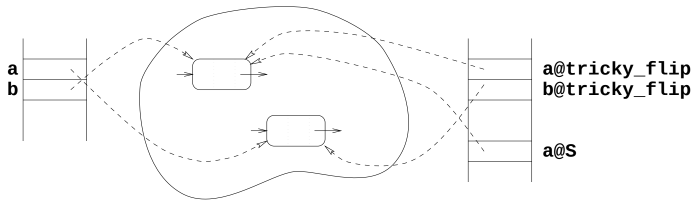
```
void tricky_flip(a,b)
Point *a; Point *b;
{ Point *tmp;
  Point *a@tricky_flip,
        *b@tricky_flip,
        *a@S; /* anchors */

   a@tricky_flip = a; /* ghost */
   b@tricky_flip = b; /* ghost */
   a@S = a;           /* ghost */
S: a->x = b->y;
   /* swap a and b */
   tmp = a; a = b; b = tmp;
T: a->y = b->x;
}
```

(a) **tricky_flip**



$$\mathrm{connect(S)} = \{(a, a@tricky\_flip), (b, b@tricky\_flip), (a, a@S), (a@S, a@tricky\_flip)\}$$

(b) program point S,



$$\mathrm{connect(T)} = \{(b, a@tricky\_flip), (a, b@tricky\_flip), (b, a@S), (a@S, a@tricky\_flip)\}$$

(c) program point T

Figure 5: Dropping Anchors and Inserting Ghost Copy Statements

points-to set at the given program point). The first two types of anchor handles are introduced to compute heap read/write sets for the entire function (in Figure 5 the handles `a@tricky_flip` and `b@tricky_flip` fall into this category). Call-site anchor handles are used to compare heap read/write sets of a function call with those of other statements. To include anchor handles in the connection analysis, ghost copy assignments are performed at function entry points and at all indirect references to the heap. Note that the points-to information about which indirect references/function calls can access the heap, enables us to reduce the number of anchor handles generated.

We use extended SSA numbers [21] to further reduce the number of anchor handles required. Although conceptually one requires a new anchor handle for each indirect reference to the heap, in fact, anchor handles can often be reused. For example, in the program in Figure 5, the anchor `a@tricky_flip` can also be used as the anchor `a@S`, because they anchor the same location: pointer `a` has not been updated between the program points the two handles are created. Thus, the same handle can be used to anchor all indirect references involving a given definition of a pointer. Our extended SSA numbering associates a new primary SSA number to a variable (including pointers), whenever it is potentially updated (including indirect updates). Thus, when we generate an anchor handle for a pointer `ptr`, we also associate its current SSA number `k` with the handle. If an anchor handle for `ptr` with the SSA number `k` already exists we do not generate a new one, and reuse the existing one for the given indirect reference too.

In subsection 2.1, we briefly illustrated how stack read/write sets are computed for function calls using *map* information from points-to analysis. Heap read/write sets for function calls, are also computed using the *map information* deposited by connection analysis, which is a context-sensitive interprocedural analysis. It generates special symbolic names to represent heap pointers which are invisible in the callee procedure, but whose connection relationships can still be modified by it.

Consider again the call to function `flip` in Figure 2. Due to parameter passing, the call generates the connection relationships: *(a,s)* and *(b,t)*. However, as the names `s` and `t` are not visible to the function `flip`, connection analysis *maps* them to special symbolic names `0+a` and `0+b`, generating the connection relationships *(a,0+a)* and *(b,0+b)*. Now the ghost copy statements at the entry to `flip` will generate the connection pairs: *(a@flip,0+a)*, *(b@flip,0+b)* in addition to the pairs *(a@flip,a)* and *(b@flip,b)*. The heap read/write sets for function `flip` in terms of the parameter anchors will be as follows:

HeapRead(flip) = {`b@flip->y, b@flip->x`}
HeapWrite(flip) = {`a@flip->x, a@flip->y`}

On *unmapping*, connection analysis has the information that: (i) `s` is mapped to `0+a`, (ii) `0+a` is connected with the anchor `a@flip`, and (iii) `a@flip->x` and `a@flip->y` are in the set HeapWrite(flip). From these facts, it deduces that in the context of function `main`, the anchored locations `s@D->x` and `s@D->y` are written by the call to `flip` at program point `D`. Similarly for pointer `t` it deduces that the locations `t@D->x` and `t@D->y` are read by the call. In our implementation, we do not generate an anchor for each argument to a call. Instead we generate just two handles, `rd_anchor@D` and `wrt_anchor@D` to respectively represent *read* and *write* anchors, and express read/write sets with respect to them, giving the following sets for the call to `flip`:

HeapRead(D) = {`rd_anchor@D->x, rd_anchor@D->y`}
HeapWrite(D) = {`wrt_anchor@D->x, wrt_anchor@D->y`}

Additionally, we perform the ghost copy assignments, `rd_anchor@D = t` and `wrt_anchor@D = s`, after the function call is processed. Due to these assignments, at statement `E` in function `main` (`E: y = s->x`), we will find `s` connected with `wrt_anchor@D`, and hence can detect that statements `D` and `E` conflict. Thus the main idea is that instead of creating an anchor for each pointer with respect to which the function call can access the heap, we just create two anchors to represent all the needed anchors. Further implementation details about anchor-augmented connection analysis can be found in [11].

2.2.4 Analysis Efficiency

We have evaluated the efficiency of our analyses with respect to a set of 16 C benchmark programs, drawn from the SPEC92, SPLASH-2 [34], Olden [25], Irvine [18] and Wisconsin [3] benchmark suites. A brief description of the benchmarks is provided in Table 1. The table also summarizes the principal data structures used by the benchmarks. Here S-Array denotes a statically allocated array, D-Array denotes a dynamically allocated array, and D-linked list denotes a doubly-linked list.

Pgm	Description	Data Structures
alvinn	SPEC92 C Program	S-Arrays
water	Molecular Dynamics	S-Arrays & Lists
health	Health Care Simul.	Quadtree & Lists
graphics	Graphics Routines	Linked Lists
circuit	Sparse Matrix Solver	D-linked Lists
nbody	N-body Simulation	Octree
specear	SPEC92 C Program	S/D-Arrays
em3d	Electromagnetics	D-Arrays & Lists
yacr2	Channel Routiner	Arrays of Structures
ks	Graph Partitioning	D-Arrays & Lists
vor	Voronoi Diagrams	Splay Tree & Lists
power	Power Optimization	k-ary Tree
nrcode2	Vector C benchmark	D-Arrays
blocks2	Comput. Biology	D-Arrays & Lists
sim	Comput. Biology	D-Arrays & Lists
eigen	Eigenvalues	D-Arrays

Table 1: Benchmark Descriptions

Pgm	NS	SR	HR	AA	PT	CT
alvinn	340	42	0	0.00	0.19	0.00
water	2829	31	569	0.53	6.90	7.90
health	392	4	116	0.55	0.51	0.66
graphics	1161	21	250	0.51	4.21	0.86
circuit	1679	10	465	0.51	0.99	2.24
nbody	1110	54	177	0.78	1.93	1.92
specear	3176	143	146	0.34	2.77	1.20
em3d	279	3	51	0.65	0.13	0.09
yacr2	3064	10	665	0.50	7.27	13.31
ks	652	44	90	0.68	0.59	0.58
vor	1623	63	479	0.50	2.87	2.59
power	741	29	151	0.09	1.05	0.15
nrcode2	312	0	53	0.19	0.16	0.05
blocks2	1149	82	279	0.51	1.66	4.30
sim	1845	69	340	0.31	1.56	2.60
eigen	296	11	44	0.30	0.10	0.05

Table 2: Analysis Statistics

In Table 2 we provide the analysis times measured on an *UltraSparc* machine in *seconds*. The points-to and connection timings (PT and CT) respectively include the additional time spent in computing stack and heap read/write sets. The table also provides the following information for each program: (i) number of SIMPLE statements (NS), (ii) number of stack and heap related indirect references (SR and HR), and (iii) the average number of anchor handles generated per indirect reference (AA). An interesting observation is that our context-sensitive pointer analyses are quite efficient for moderate size benchmarks. This indicates that if interesting sections of larger programs can be identified using linear analyses [29, 31], precise information for these sections can be obtained efficiently. We are presently experimenting with this approach to efficiently analyze large programs. Another observation is that the average number of anchor handles generated per indirect reference is about 0.50 for most of the benchmarks. This indicates that by using SSA numbers, we reduce the number of anchor handles needed by almost 50%.

3 Applications

We have used our implementation of stack and heap read/write set analyses as the basis for extending several standard scalar compiler optimizations like loop-invariant removal (LIR) and global common subexpression elimination (CSE), and for more advanced compiler applications like array dependence testing. We have also built a program-understanding tool, that displays the summary read/write information to the user via Web browsers. In the following sections we assume that we are working with the SIMPLE representation of a C program using the McCAT compiler [15].

3.1 Scalar Optimizations

We have used the algorithms from [1] to implement the LIR and CSE optimizations. Our implementation is more powerful due to improved read/write information. Further, it extends these optimizations to include *read* pointer expressions, globals and address exposed variables (variables whose address has been taken). The latter two are included because they cannot be register-promoted in the absence of alias information.

We have also implemented another optimization we call *location-invariant removal* (LcIR), which is similar in spirit to the scalar replacement technique proposed for array references [4]. Any memory reference that accesses the same memory location in all iterations of a loop is considered to be *location invariant*. For example, the pointer access `r->i` in Figure 6(a) is location invariant as the origin pointer `r` is not written inside the loop. We can replace all accesses to `r->i` with a scalar, say `tr`, as shown in Figure 6(b). To *safely* perform location invariant removal, we check that all memory references in the loop that can access the given location, are syntactically equivalent to the given location invariant reference. For example, in Figure 6(a), if `p->i` can access the same location as `r->i`, LcIR cannot be performed.

Finally, when moving a pointer reference for loop or location invariant removal, we *guard* the invariant statements with the loop condition when necessary, in order to preserve program semantics (example in Figure 7(d)). Further implementation details for the above optimizations can be found in [11].

(a)	(b)

`while (p != NULL)`
`{ S: r->i = r->i + p->i;`
`  T: p = p->next;`
`}`
	`tr = r->i;`
	`while (p != NULL)`
	`{ S: tr = tr + p->i;`
	`  T: p = p->next;`
	`}`
	`r->i = tr;`

Figure 6: Location Invariants

3.2 Experimental Results for Scalar Optimizations

In this subsection, we study the experimental results obtained by applying the above three optimizations to a suite of 12 pointer intensive C benchmarks programs briefly described in Table 1 (first 12 entries). We have collected both static (compile-time) and dynamic (runtime) statistics.

First, we present the compile-time data, which indicates the applicability of a given optimization for a program when using both the stack and heap read/write sets (Section 3.2.1). We then examine how many of these optimization opportunities are due to the presence of heap read/write sets (Section 3.2.2). Finally, we provide runtime measurements to measure the ultimate benefit of our read/write sets for decreasing the number of memory accesses, decreasing the number of instructions executed, and decreasing the running time of optimized programs (Section 3.2.3).

3.2.1 Optimizations applied with stack and heap read/write sets

Our first experimental results report how many opportunities arise for our scalar optimizations, when run with both the stack and heap read/write sets available. The purpose of this data is to demonstrate the types of optimizations possible, and to show that a significant number of opportunities arise in pointer-intensive benchmarks.

In Table 3, the three multicolumns respectively give the number of times the three optimizations: loop-invariant removal (Loop Invars), location-invariant removal (Loc Invars) and common subexpression elimination (CSE), are applied for a program. Each multicolumn is further divided into several labeled columns, which give the data for different types of expressions. The denotations for labels are as follows: (i) gl: global variables, (ii) ae: address exposed variables, (iii) addr: address calculations of the form `&a[i]` or `&(a->field)`, (iv) ind: indirect references (`*a`, `a->field`, `a[i]` where `a` is a pointer), simple array references (`a[i]` where `a` is not a pointer), and simple component references (`a.field`), and (v) expr: expressions involving computation (`lhs = op scalar`, `lhs = scalar_1 op scalar_2`). Note that location invariant optimization is not applicable for expressions involving address calculations or computation.

For several benchmarks globals and address exposed variables contribute significantly to the number of loop invariants (*water*, *specear* and *yacr2*). All these benchmarks use a number of global variables which are initialized only once in the program (at the beginning), and so are invariant for most of the loops. Our loop-invariant algorithm replaces them with temporaries which can be register allocated. Figure 7(a) shows an example from the *specear* benchmark where the global variable `Earlength` is substituted with the temporary `temp_invar_19`.

A significant number of loop-invariant address calculations are found for the benchmarks *water*, *health*, *graphics*, *nbody* and *yacr2*. They typically arise due to accesses to arrays embedded inside (heap allocated) structures. An example from the *nbody* benchmark is shown in Figure 7(b).

Invariant indirect references (*ind* column) are found in all the benchmarks (being pointer intensive). They typically arise in nested loops, where the outer loop is traversing the nodes of a recursive data structure, and the inner loop operates on the node itself. Indirect references with respect to the node in the inner loop provide opportunities for invariant detection. An example from *em3d* is shown in Figure 7(c) where we find three indirect references with respect to the pointer `nodelist`, invariant for the inner loop.

We also find significant number of *expr* invariants. It might seem that such invariants can be identified even without pointer analysis, as they only involve scalars. However, the scalars could be defined inside the loop, and these definitions may involve pointer expressions, which have to be first detected to be invariant.

We find limited applications of location-invariant removal in our benchmarks. However, they prove to be critical in the dynamic context. Opportunities for location invariant removal arise mainly due to summation/reduction operations. Typical examples include : (i) storing up the sum of all node values in the header node, and (ii) an inner loop storing the sum of its node values, to the current outer loop node. Figure 7(d) shows a loop from the *alvinn* benchmark where location invariant optimization is applied.

CSE optimization finds numerous applications in all our benchmarks, particularly for address calculations and indirect references. To give further insight, we show representative example applications of CSE from different benchmarks in Figure 8. Part(a) shows a loop from *water* where repeated array address calculations are eliminated. Part(b) shows a loop from *health*. Here the loop pointer is `list`, however each loop iteration accesses fields of the heap object pointed to by the pointer `(*list).patient`. With CSE optimization, we compute this pointer only once for each iteration, as opposed to once for every field access (as in the original program). Part(c) shows a loop from the benchmark *circuit*. Here the pointer expression `(*ch).ncolH` is common for the loop test and the statement advancing the loop pointer `ch`. With CSE optimization we compute the expression only once per iteration, as opposed to twice. Finally part(d) shows a code fragment from a loop in *vor* where a pointer expression calculated for parameter passing is reused later via CSE as the function call does not modify its value.

3.2.2 Benefits of using heap read/write sets

The data shown in Table 3 is for the case when we use both stack and heap read/write sets. In order to measure what part is due to the heap analyses, we also applied the above optimizations with only stack read/write sets. In Table 4 we compare the data for the total number of optimizations applied, for the two cases. The columns labeled as S and H respectively give the numbers for the case with only stack read/write sets being used (Stack case), and the case with both stack and heap read/write sets being used (Heap case).

Program	LIRs		LcIRs		CSE	
	S	H	S	H	S	H
alvinn	21	21	3	3	13	13
water	815	877	1	11	408	460
health	16	19	0	3	7	20
graphics	33	39	0	0	62	118
circuit	58	79	0	0	71	94
nbody	69	70	2	2	69	69
specear	89	92	13	13	140	150
em3d	4	11	0	0	3	4
yacr2	249	340	9	9	383	418
ks	47	47	8	8	52	53
vor	2	2	2	2	295	327
power	4	6	8	8	54	71

Table 4: Total Optimizations Applied

The number of loop invariants and/or common subexpressions eliminated increases moderately for the Heap case, for all the benchmarks (except for *alvinn* that does not have any heap references). The Stack case is able to detect the majority of the optimization opportunities, for two reasons. First, for the case of globals and address exposed variables, heap read/write information does not bring any added advantage. Second, if a loop or code fragment does not involve any write to *heap*, all heap-related invariants/common subexpressions for it can still be detected without a heap analysis.

The number of location invariants increases only for *water* and *health* as these are the only benchmarks in

Program	Loop Invars					Loc Invars			CSE				
	gl	ae	addr	ind	expr	gl	ae	ind	gl	ae	addr	ind	expr
alvinn	12	0	6	3	0	0	0	3	0	0	2	6	5
water	449	26	104	33	265	0	1	10	0	4	111	143	202
health	1	0	14	2	2	0	0	3	0	0	1	19	0
graphics	0	0	25	8	6	0	0	0	0	0	11	93	14
circuit	0	4	0	53	22	0	0	0	0	1	0	66	27
nbody	4	2	44	9	11	2	0	0	5	0	32	23	9
specear	25	1	2	9	55	13	0	0	0	0	2	62	86
em3d	0	0	0	9	2	0	0	0	0	0	0	2	2
yacr2	107	7	94	111	21	7	0	2	47	0	106	149	116
ks	1	5	12	19	10	1	0	7	0	0	17	26	10
vor	0	1	1	0	0	2	0	0	15	0	43	247	22
power	0	0	4	2	0	0	0	8	0	0	0	62	9

Table 3: Distribution of Optimizations Applied

```
temp_invar_18 = (*input); /* LIR */
temp_invar_19 = EarLength; /* LIR */
for (i = 0; i < temp_invar_19; i = i + 1)
{ InputState[i] = temp_invar_18;
}
```

(a) specear

```
temp_79 = (*p).un.cell.subp; /* LIR */
for (k = 0; k <= 7; k = k + 1)
{ temp_78 = temp_79[k];
  if ((temp_78 != 0))
  { temp_82 = (dsq / 4.0);
    walksub_st95(temp_78, temp_82);
  }
}
```

(b) nbody

```
while (nodelist != 0)
{
  temp_18 = (*nodelist).from_nodes; /* LIR */
  temp_19 = (*nodelist).coeffs; /* LIR */
  temp_17 = (*nodelist).from_count; /* LIR */
  for (i = 0; i < temp_17;  )
  { other_node = temp_18[i];
    coeff = temp_19[i];
    value = (*other_node).value;
    temp_21 = (*nodelist).value;
    temp_22 = (coeff * value);
    (*nodelist).value = temp_21 - temp_22;
    i = i + 1;
  }
  nodelist = (*nodelist).next
}
```

(c) em3d

```
if (sender <= end_sender) /* guard */
{
  temp_invar_2 = (*receiver); /* LcIR */
  for (  ; (sender <= end_sender) ;  )
  { temp_31 = temp_invar_2;
    temp_34 = sender;
    sender = (sender + 1);
    temp_33 = (*temp_34);
    temp_36 = weight;
    weight = (weight + 1);
    temp_35 = (*temp_36);
    temp_32 = (temp_33 * temp_35);
    temp_invar_2 = (temp_31 + temp_32);
  }
  (*receiver) = temp_invar_2;
}
```

(d) alvinn

Figure 7: Examples for Loop Invariants (LIR) and Location Invariants (LcIR)

which heap-related location invariant expressions arise, along with *ks*. However, the two heap-based location invariants from *ks* arise in loops which do not involve any other write access to *heap*, so they can already be detected without needing heap read/write sets.

3.2.3 Runtime Improvements with pointer read/write sets

As noted above, only with stack read/write sets that conservatively estimate the heap, heap-related invariants/common subexpressions may be detected in several cases. Likewise, it is also possible that an optimizing compiler that conservatively handles pointer references, could also detect many of the optimization opportunities that our analyses detect. For example, the CSE transformations shown in Figure 8(a) can also be performed without any pointer analysis information (in fact gcc does so).

In order to measure the additional benefits of our analyses over a state-of-the-art optimizing compiler, we have compared our results with the GNU C compiler [30] (gcc version 2.7.2) working at the highest level of optimization (with -O3 flag). Since our transformations are source-to-source and are performed at the SIMPLE intermediate representation, we performed the following experiment. We produced three sources for each benchmark program: (i) the dump of the SIMPLE representation of the program (*plain* version), (ii) the dump of the SIMPLE representation of the program *after* the above three optimizations are applied with only stack read/write sets (*Sopt* version), and (iii) the optimized dump with both stack and heap read/write sets being used (*Hopt* version). The SIMPLE dumps are just *simplified* C programs which can be compiled by any native C compiler.

Next, we compile all three versions with the gcc compiler with -*O3* optimization flag, and compare the *runtime* performance of the *opt* versions over the *plain* version. Note that any difference in the performance can be solely attributed to our source-to-source transformations. We collected the following run-time statistics:

- The total number of memory references made during program execution. This is an important metric as the main effect of applying the above optimizations on pointer expressions, globals and address exposed variables, should be the reduction of memory references.

- The total number of instructions executed. This reflects how many instructions could be eliminated due to loop invariant removal. Also these source-to-source optimizations can enable the compiler to be less conservative and produce better code, which can lead to a reduction in number of instructions.

- The run time of the program measured using the `/usr/bin/time` utility on an *UltraSparc* machine with only single user logged on. The run time was calculated as the sum of the system and user time reported by the `time` utility. Also the run time was averaged over three runs of the program.

We collected the first two statistics using the EEL [23] based QPT2 tool from Jim Larus, which instruments the program executable to give exact counts. However, note that run time reported is *not* from the QPT2-instrumented versions of the executables.

The comparison of the above statistics is presented in Table 5. The three multicolumns labeled "Mem Refs", "Insns" and "Run Time" respectively give the data regarding the number of memory references made, number of instructions executed and the run time. The columns labeled "Sopt" and "Hopt" in the multicolumns labeled "%Decrease" respectively give the percentage decrease achieved in number of memory references or instructions executed, by the *Sopt* and *Hopt* versions over the *plain* version. The columns labeled "Abs Decr" give the actual decrease in the number of memory references/instructions (in *millions*) achieved by the *Hopt* version over the *plain* version. Finally, in the "Run Time" multicolumn, the first column (labeled "Base Time") gives the run time in *seconds* for the *plain* version. The next two columns respectively show the percentage speedup obtained by the *Sopt* and *Hopt* versions over the *plain* version. The main observations from this table are discussed below.

The optimized versions achieve a significant reduction in the number of memory references. The highest is 35.56% for *alvinn*, while six other benchmarks achieve greater than 7% reduction. For *alvinn*, the main factor proves to be the location-invariant removal shown in Figure 7(d), that applies to three critical inner loops. For *specear*, the pointer-based array reference `state[i+1]` arises twice (on rhs) in its critical loop in function `agc`. Between the two references there is a write via another pointer-based array reference `output[i]`. Without pointer information gcc is not able to apply CSE across this write, while we can, and this brings most of the reduction.

For other benchmarks, invariants and common subexpressions spread all across the program contribute. Finally, for the *power* benchmark we actually see an increase in the number of memory references, despite the numerous applications of all optimizations (Table 4). This happens because in this benchmark some pointer expressions remain invariant through a function and are used all across it. Via CSE, all but the first occurence of this expression are substituted with a temporary. Such temporaries end up having long lifetimes, causing the register allocator to introduce spills, and perform worse than original.

```
while ( (curr_box != 0) )
{ temp_14 = (&(*curr_box).coord);
  i = temp_14[0];
  temp_15 = temp_14; /* CSE */
  j = temp_15[1];
  temp_16 = temp_14; /* CSE */
  k = temp_16[2];
  ... /* statements deleted */
  curr_box = (*curr_box).next_box;
}
```

(a) water

```
while ( (list != 0) )
{ temp_43 = (*list).patient;
  temp_44 = (*temp_43).time_left;
  if ((temp_44 == 0))
  { ... /* statements deleted */
    temp_57 = temp_43; /* CSE */
    (*temp_57).time_left = 10;
    temp_58 = temp_43; /* CSE */
    temp_59 = (*temp_58).time;
    ... /* statements deleted */
  }
  list = (*list).forward;
}
```

(b) health

```
temp_234 = (*ch).ncolH;
temp_cse_2 = temp_234;
while ( (temp_234 != 0) )
{ temp_235 = (*ch).col;
  mapUnk[temp_235] = (*ch).whichUnknown;
  ch = temp_cse_2; /* CSE */
  temp_234 = (*ch).ncolH;
  temp_cse_2 = temp_234;
}
```

(c) circuit

```
temp_62 = (*act).v1;
temp_63 = (*n).v1;
temp_61 = point_equal(temp_62, temp_63);
if ((temp_61 != 0))
{ p = (*act).v2;
}
else
{ p = temp_62; /* CSE */
}
```

(d) vor

Figure 8: Common Subexpression Elimination (CSE) Examples

Program	Mem Refs			Insns			Run Time		
	%Decrease		Abs	%Decrease		Abs	Base	%Speedup	
	Sopt	Hopt	Decr	Sopt	Hopt	Decr	Time	Sopt	Hopt
alvinn	35.56	35.56	684.92	11.64	11.64	682.04	42.70	10.30	10.30
water	15.59	15.64	388.16	4.50	4.51	407.13	64.10	5.77	6.08
health	0.00	15.41	135.27	-0.35	5.00	122.01	139.30	-1.87	1.36
graphics	12.41	14.32	236.30	7.87	9.08	236.10	42.40	1.89	3.07
circuit	10.01	13.39	67.22	1.59	1.63	34.87	54.30	0.92	1.47
nbody	8.13	8.14	81.25	-0.19	-0.18	-6.00	37.30	0.54	0.54
specear	7.49	7.49	293.27	5.64	5.64	817.16	107.00	1.78	2.90
em3d	0.83	3.30	7.95	0.08	0.23	5.87	16.40	1.22	1.22
yacr2	1.91	2.85	1.33	2.10	2.20	11.69	32.90	3.34	4.26
ks	2.01	2.01	36.64	0.71	0.71	37.09	43.90	0.91	0.91
vor	0.17	0.93	9.50	0.02	0.13	5.51	78.20	6.65	8.31
power	-0.01	-0.17	-0.41	-0.01	-0.05	-0.44	12.70	0.00	0.00

Table 5: Dynamic Improvements over *gcc -O3*

The above observations highlight the applicability of our optimizations to pointer expressions in particular. They also indicate that there may not always be a direct correlation between the number of times optimizations are applied and the actual run time improvements.

For five benchmarks, 4% to 11% reduction is achieved in the number of instructions executed. Again, for some benchmarks we see an increase instead. This happens due to pulling out invariant expressions, which either belong to an infrequently executed loop, or an infrequently executed path inside the given loop.

The percentage decrease figures are always equal or higher for the *Hopt* version compared to the *Sopt* version, with the difference being most marked for *health* and significant for *graphics*, *circuit* and *em3d*. All these benchmarks use recursive heap data structures, so heap read/write sets bring added benefits.

We see run time speedup of 10.30% for *alvinn*, 8.31% for *vor*, 6.08% for *water* and 4.26% for *yacr2*. These speedup figures are quite significant in the context of our scalar optimizations. Further they are achieved over "gcc -O3". While reduction in memory references and instructions executed, always translates into a speedup, the speedup obtained is not always in direct proportion. For example, for *yacr2* and *vor*, the percentage decrease figures are much less than for *health* or *circuit*. The reason they obtain better speedup is that our source-to-source transformations sometimes enable the native C compiler to perform better instruction scheduling due to substitution of pointer references with scalars (this happens for these benchmarks). For the same reason, even with the same instruction and memory reference counts, the *Hopt* version for *specear* achieves better speedup than the *Sopt* version.

We have also studied the effects of our optimizations in the context of parallelized programs for the EARTH-MANNA multithreaded architecture. Here pointer references mostly involve remote memory accesses. So applying LIR, LcIR and CSE to such references results in even better savings, giving upto 25% speedup [32].

3.3 Improving Array Dependence Tests

Scientific applications written in C also use arrays as principal data structures. However, unlike FORTRAN, these arrays are mostly implemented using pointers to dynamically-allocated storage. Further even statically-allocated arrays are often passed as pointer parameters. The pointer-based array references pose new problems for the array dependence tester (ADT). Consider the following simple example loop:

```
for (i = 1; i < 100; i++)
{ S: a[i] = a[i] + 1;
  T: c[i] = b[i-1] + a[i];
}
```

If the variables `a`, `b` and `c` are static integer arrays (declared as `int a[100], b[100], c[100]`), the ADT can easily identify that the loop has a flow dependence from `S` to `T`, but no loop-carried dependences. However, if these variables are declared as integer pointers which point to dynamically-allocated storage or to statically-allocated arrays, the situation becomes more complex. Now, the ADT cannot assume that two *syntactically* different array references are always independent. For example, if `a` and `b` point to the same heap object/static array, we have a loop-carried flow dependence from `S` to `T`. Using our pointer analyses, we can easily check against such possibilities. For the above loop, if either `a`, `b` and `c` point to different static arrays, or if the anchor handles `a@S`, `c@T`, and `b@T` are not connected with each other, the situation becomes identical to the static case.

We have measured the effectiveness of our pointer analyses for more precise ADT, using a set of array-based C programs (described in Table 1). For each benchmark, we collected the following ADT statistics: (i) the number of array pairs tested, (ii) the number of dependences detected, and (iii) the number of *forall* loops found using the ADT results. Clearly, one would like to eliminate as many dependence tests as possible, since each test is potentially expensive, and spurious tests may lead to spurious dependences. Reducing the number of dependences is beneficial both for better fine-grain parallelism, and for exposing more *forall* loops. More *forall* loops lead to more coarse-grain parallelism.

The data is shown in Table 6. The columns labeled P and H respectively show the numbers for ADT without any pointer information, and with both points-to and connection information. One can see a significant reduction in both array pairs tested and dependences detected. We are also able to find more forall loops for *specear*, *nrcode2*, *blocks2* and *alvinn*. These results indicate that pointer analyses can make ADT considerably more effective. In fact, some commercial compilers like *pgcc* (from Portland Group Inc) provide pragmas to get similar information (like *-Msafeptr* for the user to indicate that certain pointers do not share storage with other pointers/arrays). Finally, for the other three benchmarks, the array dependences broken fall into loops which are actually not forall loops. So we do not see an increase for them.

3.4 Program Understanding/Debugging

Our summary read/write information can also be used as a program understanding/debugging aid. For example, consider a procedure `foo (struct list *a, struct tree *b)`, with the following summary information: HeapWrite(foo) = {`a@foo->index`, `b@foo->left`, `b@foo->right`} and HeapRead(foo) = {`a@foo->index`, `a@foo->next`, `b@foo->left`,

Program	Pairs		Deps		Forall	
	P	H	P	H	P	H
specear	200	68	138	46	9	19
nrcode2	70	7	70	7	3	6
blocks2	16	13	10	7	6	9
alvinn	20	8	20	8	6	7
yacr2	28	7	28	7	23	23
sim	77	7	77	7	3	3
eigen	15	5	15	5	2	2

Table 6: Results for Array Dependence Analysis

`b@foo->right`}.

Based on this information we can make interesting observations about the effect of function `foo` on the data structures passed to it. The absence of `a@foo->next` in the HeapWrite set indicates that the function does not affect the structure of the list (does not add/delete nodes), and that it only modifies the scalar field `index` of some or all the nodes in the list (`a@foo->index` $\in$ HeapWrite(foo)).

On the contrary, the presence of `b@foo->left` and `b@foo->right` in the HeapWrite set indicates that the `left` and/or `right` pointer fields are modified for some nodes of the tree. This could imply that either new nodes have been added to the tree, or some nodes have been deleted, or some nodes have simply swapped their children. According to our experience with benchmarks, the useful information is again the negative information: which fields are not updated by the given function, with information about the *link* pointer fields being specially useful. Also, due to the hierarchical nature of our read/write sets, such information can be obtained with respect to other program constructs like loops, conditionals and function calls.

To nicely display such information to the user, we have developed a tool that uses a Web browser. We modified our c-dump utility to produce HTML version of the program, with each statement decorated with a hyperlink to a CGI script passing the *unique* statement ID as a hidden parameter. We also produced compressed files containing the pointer analysis, and read/write sets information for each statement along with its ID. We use three frames in the browser. The top frame displays the kinds of information available and the user has to click at the appropriate link to see a given flow information. This sets the the file containing the information as the current *infofile*. The left frame displays the program itself. The right frame is used as the workspace. When user clicks on a statement or a function, the CGI script (written in perl) is invoked with the statement/function ID as its argument. It looks for this ID in the current *infofile* and displays the information associated with it in a user friendly form in the right frame. The information displayed also has interesting hypertext links (clicking on a field displays the definition of its structure type). Further, clicking at a function prototype or a function call, takes one to the function body. The reader can use this tool by visiting the Web page `http://www-acaps.cs.mcgill.ca/~ghiya/info.html`.

The summary read/write information can also be used to guide data prefetching for recursive heap data structures [24], as it indicates which fields are potentially accessed with respect to a pointer, inside a function or a loop. So prefetch instructions can be placed for these fields at function/loop entry. Also one can avoid prefetching fields that are reported to be not used, thus reducing the prefetch overhead. Similarly *read only* field accesses can be considered as run time constants, which is a very useful information in a dynamic compilation context [2].

Another direct application of connection information is identification of potential memory leaks. When a heap-directed pointer `p` is updated, and no other *live* pointer is connected to it, the heap storage accessible from `p` will become inaccessible by the program. The programmer can be warned of a potential memory leak at the given statement.

4 Related Work

As summarized in the introduction, a considerable amount of work has been done on the problem of pointer analysis itself, and a detailed description can be found in [11]. In this section we concentrate on summarizing methods that use the results of pointer analysis.

Landi et al. [20] and Choi et al. [6], proposed approaches for computing side-effect information (read/write sets) in the presence of pointers. These approaches use stack-based alias analysis. With a points-to representation [10, 26, 31, 33], where all locations have names, computing read/write sets is quite straightforward and only slight modifications of standard transformations are needed as shown in section 2.1. We assume that other compilers with points-to analyses have similar applications.

More directly related to this paper are methods that use the results of heap analysis. Work in this area has been primarily focused on dependence analysis and parallelization. The important approaches include: techniques using path expressions to name locations [22], using syntax trees to name locations [14], extending k-limited graphs with location names[17]; and dependence testing based on access paths and theorem proving [18]. These approaches attempt to perform very accurate analysis, and reason about different parts of the same data structure (for example, determining if `x->left->right` possibly refers to the same location as `x->right->right` or not). We have taken a more general view of the potential uses of heap analysis, and have based our method on a more coarse-grain heap analysis that can distinguish between two data structures, but not references within the same data structure.

In terms of using the improved read/write sets from pointer analysis, for other analyses and transformations, the most relevant related work is of Wilson and Lam [33], Shapiro and Horwitz [28], and Cooper and Lu [7]. Wilson and Lam used pointer analysis results for loop parallelization. Shapiro and Horwitz study the effects of various flow-insensitive pointer analyses on the efficiency and precision of other analyses like live variable analysis and GMOD analysis, but not on actual program transformations.

Cooper and Lu study the benefits of pointer analysis in the context of register promotion. Their work focuses on promoting address exposed and global variables to registers inside loops when possible. They also describe a technique similar to location invariant removal to enregister pointer-based array references. Their empirical results also indicate a significant decrease in memory references for some programs, but no significant speedup. Their work can be considered a subset of our study as we do not focus on only loops, use a more precise heap analysis (they use malloc-site naming approach), and finally, we provide real runtime speedups over a state-of-the-art optimizing compiler as against comparing the number of operations executed collected via a simulator.

5 Conclusions and Future Work

This paper has focused on how to put pointer analysis to work. We demonstrated that the fundamental component is computing read/write sets. We briefly summarized the computation of read/sets from points-to analysis, a store-based analysis that focuses on stack-directed pointers. More importantly, we have provided a new method for computing read/write sets for connection analysis, which is a storeless heap analysis. In order to achieve this we introduced the notion of anchor handles, and read/write sets based on anchor handles.

Based on both the stack and heap read/write sets, we demonstrated a wide variety of applications. We provided a description of several scalar optimizations that can include optimizations of computations using pointers. We provided extensive static and dynamic measurements, including measuring runtime improvement due to the scalar optimizations. We also examined the effect of accurate read/write sets on array dependence testers, and outlined several other uses of read/write sets, including program understanding via a tool that interfaces with Web browsers. We believe that our results show that pointer analysis is an important part of an optimizing C compiler, and that one can achieve significant benefits from such an analysis.

Our future work will be in three major directions. Firstly, we plan to study the effect of stack and heap read/write sets on fine-grain parallelism and instruction scheduling. Secondly, we would like to compare the benefit of context-sensitive, flow-sensitive analyses (as presented in this paper) vs. flow-insensitive analyses. Finally, we plan to continue to develop new transformations for pointer-intensive programs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., Reading, Mass., corrected edition, 1988.

[2] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Penn., May 1996.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 290–301, Orlando, Flor., June 1994.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 53–65, White Plains, N. Y., June 1990.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 296–310, White Plains, N. Y., June 1990.

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, Jan. 1993.

[7] K. D. Cooper and J. Lu. Register promotion in C programs. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 308–319, Las Vegas, Nev., Jun. 1997.

[8] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. of the 1992 Intl. Conf. on Computer Languages*, pages 2–13, Oakland, Calif., Apr. 1992.

[9] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando, Flor., June 1994.

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 242–256, Orlando, Flor., June 1994.

[11] R. Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, School of Computer Science, McGill University, November 1997. In Preparation.

[12] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *Intl. J. of Parallel Programming*, 24(6), pages 547–578, 1996.

[13] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–15, St. Petersburg, Flor., Jan. 1996.

[14] V. A. Guarna, Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proc. of the 1988 Intl. Conf. on Parallel Processing*, volume II, pages 212–220, St. Charles, Ill., Aug. 1988.

[15] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in Lec. Notes in Comp. Sci., pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[16] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distrib. Systems*, 1(1):35–47, Jan. 1990.

[17] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proc. of the SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 28–40, Portland, Ore., Jun. 1989.

[18] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–229, Orlando, Flor., June 1994.

[19] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 235–248, San Francisco, Calif., Jun. 1992.

[20] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 56–67, Albuquerque, N. Mex., Jun. 1993.

[21] C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In Proceedings of CASCON'96, Toronto, Ontario, Nov. 1996.

[22] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 21–34, Atlanta, Georgia, Jun. 1988.

[23] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 291–300, La Jolla, Calif., Jun. 1995.

[24] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. of the Seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Mass., Oct. 1996.

[25] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[26] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 13–22, La Jolla, Calif., Jun. 1995.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 16–31, St. Petersburg, Flor., Jan. 1996.

[28] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 1997 Static Analysis Symposium*, Paris, France, Sep. 1997.

[29] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conf. Rec. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997.

[30] R. M. Stallman. *Using and Porting GNU CC*. Cambridge, Mass., Jun. 1992. Available via anonymous ftp from `prep.ai.mit.edu`.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 32–41, St. Petersburg, Flor., Jan. 1996.

[32] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proc. of the 1997 Conf. on Parallel Architectures and Compilation Techniques (PACT'97)*, San Francisco, Calif., Nov. 1997.

[33] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 1–12, La Jolla, Calif., Jun. 1995.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Shingh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, Jun. 1995.

[35] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, October 1996.