Online Cycle Detection and Difference Propagation for Pointer Analysis

David J. Pearce, Paul H.J. Kelly and Chris Hankin

Department of Computing, Imperial College, London SW7 2BZ, UK
{djp1,phjk,clh}@doc.ic.ac.uk

Abstract

This paper presents and evaluates a number of techniques to improve the execution time of interprocedural pointer analysis in the context of large C programs. The analysis is formulated as a graph of set constraints and solved using a worklist algorithm. Indirections lead to new constraints being added during this process.

In this work, we present a new algorithm for online cycle detection, and a difference propagation technique which records changes in a variable's solution. Effectiveness of these and other methods are evaluated experimentally using nine common 'C' programs ranging between 1000 to 55000 lines of code.

1 Introduction

Pointer analysis is the problem of determining beforehand what the pointer variables in a program may target. Any algorithm for doing this will always be approximate and the aim is to produce the most accurate (smallest) solution possible, for each variable, in practical amounts of time and space. A solution is regarded as valid if all actual targets are included, although there may also be extra *spurious* targets.

This work is about improving the runtime of such analyses, in particular those employing a worklist algorithm. Our main contributions are:

- An original algorithm for online cycle detection.
- A difference-propagation solver for the pointer analysis problem. This reduces work by propagating only the change in solution for a variable, rather than the whole solution.
- Empirical data comparing these and other techniques.

In this paper, as is commonly the case, we only consider flow- and context-insensitive analyses.

1.1 Flow- and Context-Sensitivity

A common way of categorising this area is in terms of *flow-* and *context-sensitivity*. The former indicates whether statement order should be considered. Thus, in the following, a flow-insensitive analysis would conclude that x can *point to* both y and w:

```
(1) z = &y;
(2) x = z;
(3) z = &w;
```

This *conservative* result arises as the algorithm is unaware of the ordering between statements. Thus, it reasons that (3) could be executed before (2) and vice-versa. The advantage of doing this comes from a reduced space requirement, as it is no longer necessary to store separate solutions for each variable at different program points.

In a similar fashion, a context-insensitive analysis ignores *calling context*. To see this more clearly, consider the following:

```
       int *simple(int *q) {return q;}
(C1)   x = simple(&a);
(C2)   y = simple(&b);
```

In this case, there are two calling contexts, $C1$ and $C2$. A context-insensitive analysis merges these into one, which could be thought of as replacing $C1$ and $C2$ with:

```
(C1+2) {x,y} = simple({&a,&b});
```

Which means: *analyse simple as though* `q` *points to both* `a` *and* `b` *and assign the result to both* `x` *and* `y`. Thus, such an analysis would conclude that `x` and `y` can point to `a` and `b`. Again, the reasoning behind this seemingly wasteful simplification is practicality: If this was not done then an analyser would effectively be inlining every function and this is known to be unscalable.

1.2 Organisation

The remainder of this paper is organised as follows: Section 2 will cover related work and any background neces-

```
char *f(char *p) {
 char *q;
 if(...) q = ``bar''; /* $f_q \supseteq \{t1\}$ */
 else q = p;          /* $f_q \supseteq f_p$ */
 return q;            /* $f_* \supseteq f_q$ */
}

void main(void) {
 char *r,*s,*p,**t,**q;
 p = ``foo'';        /* $main_p \supseteq \{t2\}$ */
 if(...) r = f(p);   /* $f_p \supseteq main_p, main_r \supseteq f_*$ */
 else r = p;         /* $main_r \supseteq main_p$ */
 s = r;              /* $main_s \supseteq main_r$ */
 r = ``test'';       /* $main_r \supseteq \{t3\}$ */
 q = &r;             /* $main_q \supseteq \{main_r\}$ */
 t = q;              /* $main_t \supseteq main_q$ */
 *t = s;             /* $*main_t \supseteq main_s$ */
}
```

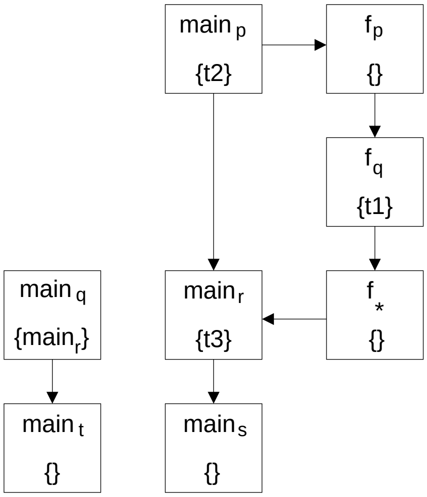


Figure 1. An artificial example illustrating how the initial constraint graph is derived from the source code. The "tX" constants represent the string objects and f_* the return value of f. Roughly speaking the graph can be solved by propagating the solution for a node to all those reachable from it. However, there is a complex constraint $*main_t \supseteq main_s$, which cannot be directly represented in the graph. But, it must eventually lead to an edge from $main_s$ to $main_r$ being added. This will happen during solving, sometime after $main_r$ has been propagated into the solution of $main_t$.

sary. Section 3 will overview three methods for improving the runtime of such analyses. These will then be empirically evaluated and discussed in Section 4. Finally, conclusions will be drawn and future work considered in Section 5.

2 Background

Flow- and context-insensitive pointer analysis has been studied extensively in the literature (see e.g. [10, 14, 21, 6, 23]). These works can, for the most part, be placed into two camps: extensions of either Andersen's [6] or Steensgaard's [23, 22] algorithm. The former use *inclusion constraints* and are more precise but slower. The latter adopt *unification systems* and sacrifice precision in favour of speed.

We will now examine each method in turn, paying particular attention to inclusion constraints as our work is in this area.

2.1 Inclusion Constraints

This general approach to pointer analysis, first suggested by Andersen, comes under the banner of *set-based constraint solving* (see e.g. [3, 4]). This involves generating and solving simple set-constraints, which are often referred to as inclusion constraints due to their use of the $\supseteq$ operator.

A small language is used for this purpose, where the domain of variables is denoted by VAR. In addition, those whose addresses have been taken are members of $VAR_\& \subseteq VAR$. Thus, the constraints themselves take the form:

$$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid \ *p \supseteq q$$

Where p and q are variables from VAR and $*$ is the usual 'C' dereference operator. Those involving the dereference operator are termed *complex constraints*. Finally, a solution to a constraint set is an assignment to each variable from $\mathcal{P}(VAR_\&)$, such that all constraints are satisfied.

2.1.1 Constraint Graph Formulation

In his original formulation, Andersen simply maintained the constraints in a vector. However, a more suitable representation is a *constraint graph*. This was first used by Heintz and Tardieu [14] and, although our description varies slightly, can be constructed as follows:

1. For every variable p, a unique vertex v_p is created.
2. An edge $v_p \leftarrow v_q$ is added for each constraint $p \supseteq q$.
3. Each vertex v_p is associated with a solution set $Sol(v_p) \supseteq VAR_\&$. This is initialised with all variables q involved in a constraint $p \supseteq \{q\}$.

In what follows, we often use a variable and its vertex interchangeably when the meaning is clear.

The constraint graph can be thought of as a dependence graph composed with a solution set for each vertex. Furthermore, as there is no clear means of expressing the *complex constraints*, we simply assume that they remain close-at-hand. This is to permit, for a given variable, quick iteration through those constraints which dereference it. Figure 1 provides a sample translation from 'C' code into the initial constraint graph.

2.1.2 Solving

At this point, the constraints can be solved by repeatedly selecting an edge $v_x \rightarrow v_y$ and merging $Sol(x)$ into $Sol(y)$ until a fixpoint is reached. This is often referred to as *convergence*. During this process, new edges arising from the complex constraints must be added to the graph. To see why this is so, consider the complex constraint $p \supseteq *q$. Suppose that initially $Sol(q) = \emptyset$, but at some point during the analysis $Sol(q) = \{x\}$. Clearly, then, there is a dependence from x to p and, furthermore, this could not have been known at graph construction time. Therefore, the edge $x \rightarrow p$ must be added as the solution for q becomes available.

The choices of which edge to select and when to process a complex constraint are important factors affecting convergence time. These issues were not addressed by Andersen, who used a simple scheme where constraints are processed in turn. As we shall see, much more sophisticated algorithms are possible.

The classical solution to this type of problem is the worklist algorithm (see e.g. [19]). Such an algorithm operates by initially placing all nodes onto a worklist. Then a node is chosen from the list and its solution propagated along all outgoing edges. Any successors whose solution has now changed are placed onto the worklist. This continues until a fixpoint is reached. Generally speaking, these algorithms are assumed to be working on a static graph. Thus, we must extend them to deal with the dynamic setting caused by the complex constraints. The general idea is to process those constraints involving $*p$ as soon as $Sol(p)$ changes. Good places to do this are when p is taken off the worklist or when it is put on. Figure 2 provides an example worklist solver using the former.

The remaining issue is *worklist selection strategy*. This is, in part, the subject of this paper and, although a large amount of work has been done in the static setting (see e.g. [5, 15, 7, 8, 16]), there appears to have been little for the dynamic case [17, 12]. In Section 3.1 we return to this.

```
procedure solve()
 $W = V$;

 while $|W| > 0$ do
    $n$ = select($W$);

    // process constraints involving $*n$
    foreach $c \in C(n)$ do
      case $c$ of
      $*n \supseteq w$:
        foreach $k \in Sol(n)$ do
          if $w \rightarrow k \notin E$ do
            $E \cup= w \rightarrow k$;
            $Sol(k) \cup= Sol(w)$;
            if $Sol(k)$ changed then $W \cup= \{k\}$;
      $w \supseteq *n$:
        foreach $k \in Sol(n)$ do
          if $k \rightarrow w \notin E$ do
            $E \cup= k \rightarrow w$;
            $Sol(w) \cup= Sol(k)$;
        if $Sol(w)$ changed then $W \cup= \{w\}$;

    // propagate solution to successors of n
    foreach $n \rightarrow w \in E$ do
      $Sol(w) \cup= Sol(n)$;
      if $Sol(w)$ changed then $W \cup= \{w\}$;
// end while
```

Figure 2. The basic worklist constraint solver. The algorithm assumes that Sol has been initialised with all trivial constraints of the form $p \supseteq \{q\}$. The set $C(n)$ contains all complex constraints involving "$*n$". Selecting a node automatically removes it from the worklist.

Another interesting approach to inclusion-based constraint solving can be found in the work of Heintz and Tardieu [14]. Their approach, roughly speaking, is to repeatedly recompute the solution for each dereferenced variable n until no change is observed. This is achieved by performing a reverse depth-first search starting at v_n, which searches out all variables contributing to $Sol(n)$ and combines their solutions into it. New edges arising from constraints involving "$*n$" are added as soon as $Sol(n)$ has been recomputed. Completing this provides only the solutions for dereferenced variables. This may be sufficient or, alternatively, a final phase could be employed to solve the (now static) constraint graph. Their work provides some evidence that sizeable programs ($\geq$ 440KLOC) can be analysed in a matter of seconds.

In addition to the above, there have been a number of other ideas put forward for improving convergence time and space usage. The most notable being *variable substitution* [21, 10]. This idea arises from the observation that variables must often have the same solution. Thus, space can be saved by representing them with a single vertex and/or solution set. The clearest example of this arises with variables involved in a cycle. As the constraint graph is dynamic in nature, full cycle detection requires an online algorithm. In [10] such an algorithm, albeit rather crude, is applied to constraint solving and significant speedups are observed. In Section 3.2, we present a better solution for detecting cycles online.

The algorithm of Heintz and Tardieu also uses an online cycle detector. However, they effectively get this for free as a byproduct of the reverse depth-first search.

2.2 Unification Algorithms

The algorithms presented by Steensgaard [23, 22] were the first example of a unification-based approach to pointer analysis. The idea is to enforce the invariant that, for each variable x, $|Sol(x)| \leq 1$. This reduces the space required to hold the solution from $O(n^2)$ to $O(n)$ and, with some clever trickery, a near-linear time complexity is achieved.

However, these improvements come at the expense of precision, which can be explained by comparing how simple assignments, such as $x = y$, are dealt with. An inclusion-based system, such as those discussed previously, says that the solution of x must include that of y. A unification system, however, states that the solution of x must *equal* that of y. The following attempts to clarify what this really means:

```
x=&z;
x=y;
```

In the above, the only way for $Sol(x) = Sol(y)$ to hold is if $z \in Sol(y)$. In other words, we *unify* the solutions of x and y, resulting in the conservative conclusion that y could point to z.

Much work has been done on this approach to improve the overall precision (see e.g. [9, 11]) and there would appear to be some indications that the actual loss compared with an inclusion-based system is quite small. However, in the face of results indicating that Andersen's algorithm is scalable, such as those of Heintz and Tardieu, the future for unification seems unclear.

3 Convergence Techniques

In this section we examine three specific techniques for improving the convergence time of an inclusion-based constraint solver: *iteration order, cycle detection* and *difference propagation*.

3.1 Iteration Order

As mentioned in Section 2.1.2, an integral part of a worklist algorithm is the strategy for choosing which node to process next. This is often referred to as the *iteration order*. The problem, then, is that selecting the wrong node can result in extra work. Figure 3 illustrates this.

For a static graph, iterating in topological order or *reverse post-order* is a good approach. This is achieved by maintaining two priority queues, $current$ and $next$, of vertices with priority given to those earlier in the r.p.o. The plan now is simple: nodes are placed onto the worklist by loading them into $next$ and are selected by taking from $current$. If $current$ is empty it is reloaded from $next$, which is then emptied. Thus, the worklist is empty only when both queues are empty. The reader is referred to [19] for a more detailed description of this. There are many variations and improvements possible. For example, it is possible to place nodes directly into $current$, if they come higher in the r.p.o than that currently being visited. Also, using just a single priority queue can be advantageous in some cases.

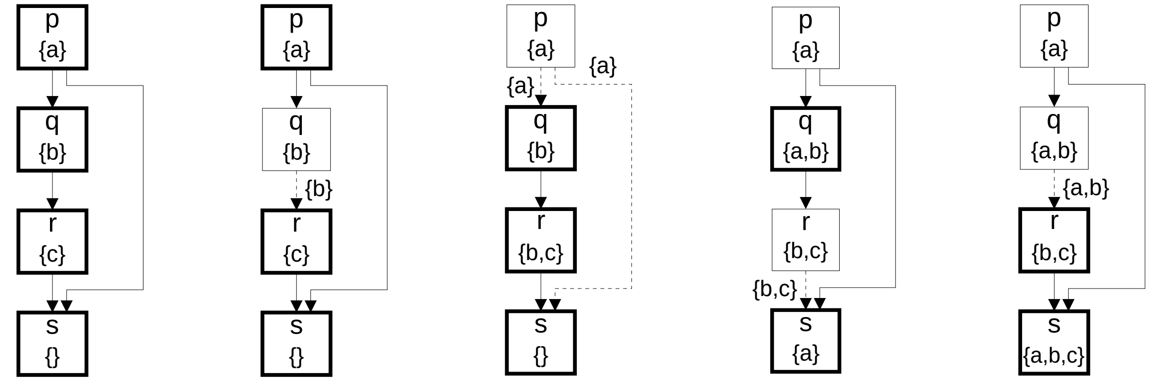
In general, the problem with this approach is the handling of cycles. The solution is to identify these *strongly connected components* and iterate each until completed before moving on. This was first suggested in [15] and a good examination can also be found in [19].

At this point, we turn our attention to the constraint graph. As we know, this is a dynamic graph and, therefore, we cannot completely determine beforehand the topological order or strong components. Furthermore, even if we used online algorithms to do this the approach used previously would not be optimal. Figure 4 demonstrates why. Nevertheless, one may still suppose that processing the graph in topological order will perform well. Indeed, so long as nodes are visited fairly, it does. But, experiments we have conducted suggest a simpler scheme, known as *least recently fired* [17], is just as effective. The idea is to prioritise nodes by when they were last visited, so that a node is chosen over another if it was visited less recently. In Section 4, we compare the LRF scheme with two simple and oft-used strategies: LIFO and FIFO.

3.2 Online Cycle Detection

As discussed in Section 2.1.2, cycle detection is a useful component for an inclusion-based constraint solver. In this section, we now present a powerful and original *online* solution to this problem, based on the work of Marchetti-Spaccamela *et al.* [18].

The pseudo-code is listed in Figure 5. The algorithm operates by maintaining a topological ordering of the vertices, which is represented by the $n2i$ and $i2n$ arrays. An invari-



(a) Initially, all nodes are on the work list.

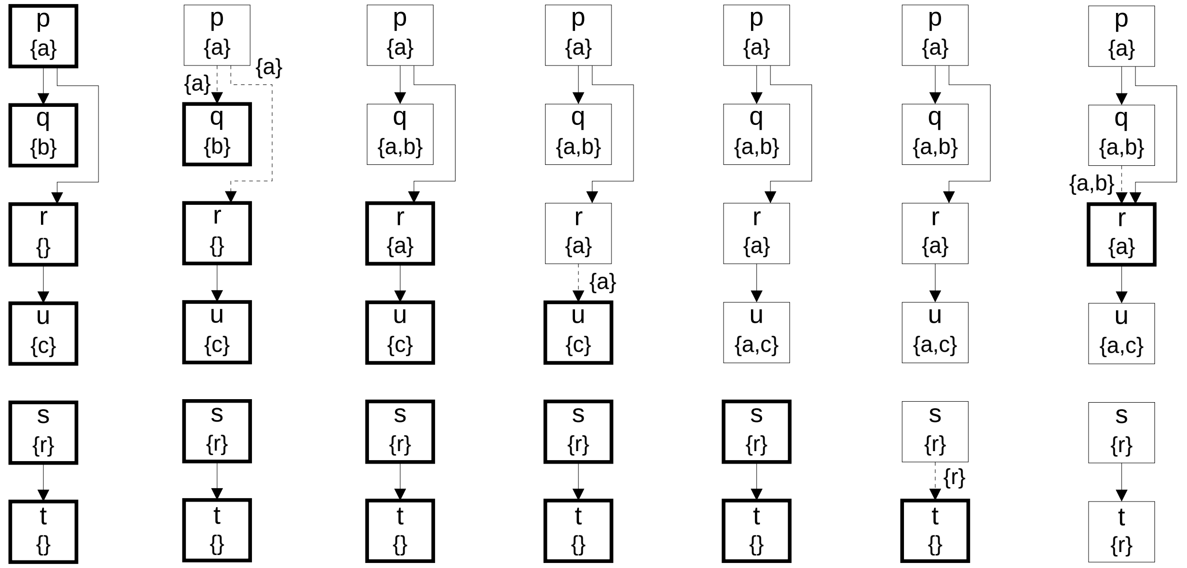
(b) Vertex q is selected first.

(c) Now vertex p, causing q to be placed back onto the worklist.

(d) Next vertex r is visited.

(e) Vertex q is now revisited.

Figure 3. Solving a simple constraint graph, using an arbitrary selection strategy. The initial graph is shown on the left. A bold border indicates that the vertex currently resides on the worklist. The dashed edges indicate where propagation occurs. Note, the algorithm hasn't finished at part e, because r and s must still be selected. The point is that some nodes are visited twice when they don't need to be. The optimal selection order is p>q>r>s.



(a) (b) (c) (d) (e) (f) (g)

Figure 4. The diagrams show a constraint graph (drawn topologically) being converged using an online topological selection strategy. We assume that the constraint set included a single complex constraint, $*t \supseteq q$. In diagram (g), node t is processed, causing the edge $q \to r$ to be added. Furthermore, the algorithm immediately propagates across this edge, causing r to be put back on the worklist. Thus, nodes r and u will be revisited after (g). The point is that leaving s and t until last means the new edge is not discovered soon enough. Had they been visited earlier, each node could have been visited just once. Thus, an online topological strategy is not optimal.

ant is enforced which states that if $x \rightarrow y \in E$ then y comes after x in the order ($n2i[y] > n2i[x]$). Thus, when an edge $x \rightarrow y$ is inserted there are two cases to consider:

1. $n2i[y] \geq n2i[x]$ - The two vertices are already ordered correctly and we do nothing.

2. $n2i[y] < n2i[x]$ - Vertex y is positioned before x in the ordering. To resolve this a depth-first search is performed starting from y, limited to nodes between y and x in the order. This uncovers vertices reachable from y which should now come after x. These are then shifted past x in the ordering. If x is reached during the search then a cycle has been detected and we back propagate this information to uncover those nodes involved.

The algorithm achieves an amortised cost over $\Theta(E)$ insertions of $O(V)$, which is a good improvement upon the $O(V + E)$ complexity of the offline algorithm. A proof of this can be found in [18]. The reader is referred to [20] for a more detailed examination of this algorithm and its complexity.

3.3 Difference Propagation

Difference propagation is a technique first suggested by Fecht and Seidl [13]. They proposed a general framework for applying it to distributive constraint systems. However, this is unable to describe constraints which have a dereferenced variable on the left hand side. Thus, the algorithm we provide here is really an instance of their framework extended to cope with the constraint system of Section 2.1.

The rough aim of the technique is to reduce the cost of propagating the solution for a node to its successors. In a standard worklist solver (see e.g. Figure 2), propagation along an edge $x \rightarrow y$ occurs by merging $Sol(x)$ into $Sol(y)$. This operation is likely to be linear in the size of the smaller set and, therefore, reducing the size of sets involved should yield an improvement.

The key idea, then, is realising that each element of $Sol(x)$ only needs to be propagated along an edge once. Figure 6 attempts to clarify this. Clearly, for this to work a sufficient number of nodes must be visited more than once and there are three likely reasons why this can happen: *poor iteration order*, *complex constraints* and *cycles*. Figures 3 and 4 provide examples of the first two.

The new solver is given in Figure 7. A key component is the difference set, $\Delta(n)$, which contains the *approximate* change in solution for each node. We can think of $\Delta(n)$ as the collection point for elements propagated to n. It is approximate as it may contain values which are already in $Sol(n)$. Each time n is visited the algorithm computes δ, the *actual* change in solution, by taking the difference between $\Delta(n)$ and $Sol(n)$. This is likely to be smaller than the solution itself and can, therefore, save unnecessary work when processing complex constraints and in propagating. The complex constraints themselves must be handled with care because, when a new edge $x \rightarrow y$ is added, we must ensure all values in $Sol(x)$ make their way into $Sol(y)$. This is achieved by propagating $Sol(x)$, not $\Delta(x)$, into $\Delta(y)$.

```
procedure add_edge($t \rightarrow h$)
  $lb = n2i[h];\ ub = n2i[t];$
  if $lb < ub$ then
    mark $t$ as $in\_component$;
    dfs($h$); shift();

procedure dfs($n$)
  mark $n$ as $visited$;
  forall $n \rightarrow w \in E$ do
    if $n2i[w] \leq ub$ then
      if $w$ unmarked do dfs($w$);
      if $w$ marked $in\_component$ then
        mark $n$ as $in\_component$;

procedure shift()
  unmark $t$; $shift = 0$;
  for $i = lb$ to $ub$ do
    $n = i2n[i]$;
    if $n$ marked then
      if $n$ marked $visited$ then push($n, L$);
      else push($n, C$);
      $shift = shift+1$;
      unmark $n$;
    else allocate($n, i-shift$);
  // place visited nodes after t in ordering
  for $j = 0$ to $|L|$ do
    allocate($L[j], i-shift$); $i = i+1$;
  // check if new cycle detected
  if $|C| > 0$ then cycle_detected($t \cup C$);

procedure allocate($n, i$)
  // assign n to topological index i
  $n2i[n] = i;\ i2n[i] = n;$
```

Figure 5. Our algorithm for online cycle detection. The $i2n$ and $n2i$ arrays are the [topological-]index-to-node and node-to-[topological-]index maps respectively.

The algorithm is likely to visit more nodes than the standard solver. The reason being that it places a node n onto the worklist when $\Delta(n)$, not $Sol(n)$, has changed. Thus, it is now possible for a node n to be placed onto the worklist as a result of some element, already present in $Sol(n)$, being inserted into $\Delta(n)$. We present some experimental data in Section 4 which attempts to quantify the effect of this.

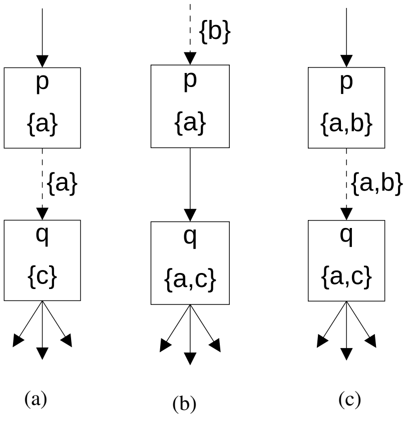


Figure 6. Illustrating unnecessary work performed by the standard worklist solver. The diagrams show part of a constraint graph during convergence. We see an initial propagation from p to q occurring (a). The solution for p is later updated (b) and this is eventually propagated to q (c). The point is that "a" does not need to be repropagated in (c). In fact, only the change in p's solution does, which in this case is $\{b\}$.

4 Experimental Study

In this section we provide empirical data, over a range of benchmarks, on the runtimes of worklist solvers using different combinations of the techniques described in Section 3. The purpose is to facilitate an understanding of how effective these methods can be.

Table 1 provides information on our benchmark suite. All are part of the GNU system and, as a result, their full source can be obtained from `http://www.gnu.org`. The SUIF 2.0 research compiler from Stanford [2] was deployed as the frontend for generating constraint sets. In all cases, we were able to compile the benchmarks with only superficial modifications, such as adding extra "`#include`" directives for missing standard library headers.

The constraint generator operates on the full 'C' language and a few points must be made about this:

- *Heap model* - A single heap object per static allocation site was used.
- *Structs* - All elements of a structure were mapped to a single constraint variable.
- *Arrays* - Treated in a similar fashion to structs, by ignoring the index expression.

```
procedure solve()
 foreach $n \in V$ do
      $W \cup= \{n\};\ \Delta(n) = Sol(n);\ Sol(n) = \emptyset;$

 while $|W| > 0$ do
      $n$ = select($W$);
      // compute actual change in solution
      $\delta = \Delta(n) - Sol(n);$
      $Sol(n) \cup= \delta;\ \Delta(n) = \emptyset;$

      // process constraints involving $*n$
      foreach $c \in C(n)$ do
         case $c$ of
         $*n \supseteq w$:
            foreach $k \in \delta$ do
              if $w \rightarrow k \notin E$ do
                 $E \cup= w \rightarrow k;$
                 $\Delta(k) \cup= Sol(w);$
                 if $\Delta(k)$ changed then $W \cup= \{k\};$
         $w \supseteq *n$:
            foreach $k \in \delta$ do
              if $k \rightarrow w \notin E$ do
                 $E \cup= k \rightarrow w;$
                 $\Delta(w) \cup= Sol(k);$
            if $\Delta(w)$ changed then $W \cup= \{w\};$

      // propagate $\delta$ to successors of $n$
      foreach $n \rightarrow w \in E$ do
         $\Delta(w) \cup= \delta;$
         if $\Delta(w)$ changed then $W \cup= \{w\};$
 // end while
```

Figure 7. The Difference propagating worklist solver. Sol and C are initialised the same as for Figure 2.

- *String Constants* - These were preserved intact and not ignored or combined into a single object.
- *Indirect Calls* - Indirect calls were handled using a mechanism similar to processing the complex constraints. Exact details are, unfortunately, beyond the scope of this paper.
- *External Library Functions* - These, almost entirely, came from the GNU C library and were modelled using hand crafted summary functions, which captured only aspects relevant to pointer analysis.

The results, listed in Tables 2 and 3, were generated on a 900Mhz Athlon based machine with 1Gb of main memory, running Redhat 8.0 (Pysche). The executables were compiled using gcc 3.2, with optimisation level "-O2". Timing

Benchmark		L.O.C.	Constraints Set					# Cycles		Avg Set
Name	Version		Triv	Simp	Comp	Added	#Var	Initial	Final	Size
bool	0.11	3246	138	796	266	3001	1031	26/(190)	17/(294)	7.43
time	1.7	1244	92	530	11	2	848	5/(16)	5/(16)	0.44
bc	1.06	7079	396	1324	358	2468	1609	53/(208)	52/(342)	5.03
chess	5.02	7451	953	851	27	23	1658	15/(139)	17/(150)	1.24
grep	2.0	7243	327	1937	501	552	2298	49/(349)	45/(414)	1.48
make	3.79.1	16164	1019	4030	1320	20617	4105	103/(595)	89/(837)	33.07
uucp	1.06.1	10256	625	2595	586	14970	2994	46/(250)	41/(505)	17.8
gawk	3.1.0	19598	1320	7054	1511	280972	6542	110/(1422)	86/(2311)	221.45
bash	2.05	55324	2813	11636	2365	240331	10649	211/(1203)	201/(2154)	111.85

Table 1. The Benchmark Suite. L.O.C. reports non-blank, non-comment lines only. Constraints are counted as Trivial ($p \supseteq \{a\}$), simple ($p \supseteq q$), Complex (involving '$*$') or Added [during convergence]. The number of cycles in the initial and final constraint graphs are provided, along with the total number of variables involved in a cycle (shown in brackets). The last column provides the average size of a target set in the solution.

	bool	time	bc	chess	grep	make	uucp	gawk	bash
WL	0.0205	0.000364	0.0216	0.00123	0.00518	0.609	1.07	656.0	207.0
WF	0.0156	0.000366	0.0146	0.00127	0.00579	0.627	0.833	190.0	46.4
WR	0.0168	0.000427	0.0183	0.00179	0.00609	0.465	0.465	58.6	49.4
WDL	0.0241	0.000537	0.0245	0.00227	0.00729	0.436	0.403	71.1	33.1
WDF	0.019	0.000509	0.0191	0.00227	0.00655	0.386	0.327	41.1	17.7
WDR	0.019	0.000604	0.023	0.00253	0.00767	0.41	0.26	37.6	26.4
WSL	0.0188	0.000326	0.0194	0.000955	0.00446	0.564	1.08	566.0	199.0
WSF	0.0135	0.000338	0.0124	0.000998	0.00456	0.449	0.809	207.0	56.3
WSR	0.014	0.00039	0.0169	0.00119	0.00461	0.329	0.283	44.7	34.9
WDSL	0.0193	0.000456	0.0206	0.00225	0.00574	0.394	0.383	54.9	36.7
WDSF	0.0152	0.000504	0.0173	0.00271	0.00576	0.309	0.3	34.1	17.2
WDSR	0.0157	0.000522	0.0207	0.00209	0.0064	0.336	0.214	31.1	19.7
WCL	0.105	0.000347	0.0542	0.00148	0.0238	1.55	0.616	166.0	87.9
WCF	0.0459	0.000381	0.0456	0.00187	0.0241	0.666	0.558	150.0	34.7
WCR	0.0313	0.00081	0.041	0.00165	0.023	0.367	0.16	5.14	7.05
WDCL	0.0379	0.000506	0.0402	0.00242	0.0254	1.0	0.337	29.0	23.2
WDCF	0.0246	0.000517	0.0359	0.0025	0.025	0.714	0.229	11.3	12.7
WDCR	0.0341	0.000534	0.0482	0.00256	0.0257	0.437	0.182	5.85	7.09

Table 2. Empirical data showing the effects of various convergence techniques. The various algorithms are on the left hand side and the key is: W=Worklist, S=Static cycle detection, C=online Cycle detection, D=Difference propagation and L,F,R indicate a LIFO, FIFO and LRF worklist selection strategy respectively. Section 3 details the specifics of each technique. The data was averaged over five runs with a very low variance being observed.

	WL	WF	WR	WDL	WDF	WDR	WCL	WCR	WDCL	WDCR
bool	2302	2176	2057	3876	2524	2349	1573	1338	2401	1372
bc	4499	3373	3225	8357	4393	3711	3369	2141	5583	2641
chess	2189	2191	1896	2394	2241	1947	1951	1662	2016	1691
grep	4425	4274	3644	5645	4774	3857	3628	2874	4385	3086
make	16615	15194	12904	45293	20287	14270	11364	6512	35795	7138
gawk	140440	36261	22805	5899208	96296	25230	58790	9289	119725	10007
bash	231507	46385	43131	2173446	82963	45813	156996	21534	310919	23583

Table 3. The values are the number of times any node was taken off the worklist. This is the visit count. Space alone has prevented us from showing data for all benchmarks and algorithms

was performed using the `gettimeofday` function and the implementation was in C++, making extensive use of the *Standard Template Library* and *Boost Library* [1].

4.1 Discussion

We will now point out key features of the data, along with some remarks. In doing this, it is helpful to split the benchmarks into two categories: the small ones (`bool`, `time`, `bc`, `grep` and `chess`) and the large ones (`make`, `uucp`, `gawk` and `bash`). Also, note that we regard WL, WF and WR as the baseline solvers.

Selection Strategy - Looking at Table 2 and the baseline algorithms WL, WF and WR we observe that LRF is never optimal for the small benchmarks, but is generally best for the large ones. This trend is repeated throughout the WD, WS and WDS families and, in the WC and WDC categories, we see LRF gaining a significant advantage over LIFO and FIFO. Table 3 suggests that LRF always has a lower visit count than the other strategies. However, we note that this reduction is not always significant for the smaller benchmarks.

Comments: The LRF scheme requires the use of a priority queue, giving it a higher overhead. Hence, it seems reasonable to conclude that this can outweigh any small saving in visit count.

Cycle Detection - From the data in Table 2, we see that the WS algorithms almost always beat their baseline counterparts. We also see from Table 2 that online detection does not perform well on the small benchmarks. However, WCR is usually better than WSR on the large ones. In particular, WCR and WDCR are by far the fastest solvers for the two largest benchmarks, `gawk` and `bash`. Table 3 indicates that online cycle detection dramatically reduces the visit count, irrespective of benchmark size.

Comments: Online cycle detection is expensive and thus a high return is needed to show any improvement in runtime. Looking at visit count alone is not sufficient to explain why it only pays off on the larger benchmarks. We must also consider the relative cost of visiting a node and the final column of Table 1 attempts to measure this. The figures show only the average set size once convergence is complete, but we believe this gives an indication of the set sizes involved during convergence. Thus, in Table 1 we observe that the larger benchmarks have a significantly greater average set size. We feel this implies the cost per visit to be relatively bigger for the larger benchmarks and, hence, more is gained by reducing the number of visits.

The performance of the online cycle detector, in general, is pleasing as it demonstrates that the technique can be efficient and beneficial over static detection.

Difference Propagation - For the large benchmarks, the results in Table 2 show that difference propagation is always an improvement over the baseline. The exact opposite holds for the small benchmarks. We note with curiosity that, for the two largest benchmarks, WDL and WDF go significantly faster than WCL and WCF, but that WDR is much slower than WCR. We also see that adding difference propagation to the WS family almost always yields something better for the large benchmarks, but that WDCR is worse than WCR in all but one. Finally, Table 3 shows that difference propagation causes the expected increase in visit count.

Comments: Again, this technique introduces an overhead which appears unjustified on the smaller benchmarks. However, it does appear to show promise, although it remains unclear why the WDCR algorithm does not do better. The data for `bash` shows a sizeable increase in visit count, with only a small drop in performance. This suggests that visit count may not be relevant in explaining the problem. And so, we can only speculate that with cycles removed, there are simply not enough nodes being visited more than once. This may be supported by the data from Table 1, which indicates that the final constraint graphs have a high outdegree and thus will propagate their solution to many nodes in a single visit.

5 Conclusion

We have explored the use of some existing and some original methods for improving the convergence time of inclusion-constraint solvers in the field of pointer analysis. Our results indicate that worklist selection strategy is important, that online cycle detection is feasible and effective and, finally, that difference propagation has potential.

In the future we hope to look at larger benchmarks, investigate the difference propagation technique further, add an implementation of the Heintz-Tardieu solver and introduce flow-sensitivity.

6 Acknowledgements

David J. Pearce is supported by a PhD studentship from the U.K. Engineering and Physical Sciences Research Council (EPSRC).

References

[1] Boost C++ libraries, http://www.boost.org.

[2] The SUIF 2 research compiler, Stanford University, http://suif.stanford.edu.

[3] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, Nov. 1999.

[4] A. Aiken and E. L. Wimmers. Solving systems of set constraints. In *Proc. 7th Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[5] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Conference on FormalMethodsinProgrammingandtheirApplications*,pages128–141,1993.

[8] L.-L. Chen and W. L. Harrison. An efficient approach to computing fixpoints for complex program analysis. In *Proc. ACM Conference on Supercomputing*, pages 98–106, 1994.

[9] M. Das. Unification-based pointer analysis with directional assignments. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[10] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[11] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[12] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proc. Static Analysis Symposium*, pages 189–204, 1996.

[13] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proc. European Symposium on Programming*, pages 90–104, 1998.

[14] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[15] S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[16] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.

[17] A. Kanamori and D. Weise. Worklist management strategies. Technical Report MSR-TR-94-12, Microsoft Research, 1994.

[18] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.

[19] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[20] D. J. Pearce and P. H. J. Kelly. Online algorithms for maintaining the topological order of a directed acyclic graph (work in progess, available upon request). Technical report, Imperial College, 2003.

[21] A. Rouhtev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[22] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proc. 6th International Conference on Compiler Construction*, pages 136–150, 1996.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.